CHAPTER 13

Building ActiveX and COM+ Components Advanced Topics

In This Chapter

Writing a simple MFC-based ActiveX control

Writing a simple ATL-based ActiveX control

Creating application test procedures for both MFC- and ATL-based ActiveX controls

ActiveX components form the basis of most applications today. No matter what kind of application you want to create, it almost always will include an ActiveX control of some type. In Chapter 12, I discussed the theory behind ActiveX as well as some of the problems you might encounter when working with it. This chapter puts the information that you learned in the previous into action.

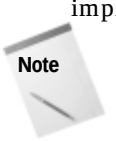
The first section of this chapter is where we'll finally look at some examples of how you can create your own components. In fact, we'll create the same component using two different techniques so that you can see the differences between them. You'll be surprised to find that even though the two components act precisely the same, there are a lot of differences between them in performance, development time, and size. These differences will show why you need to choose your method of development carefully from the outset.

Once you see the two components we've created, you'll want to test them. The next section of the chapter shows you how to test the components on the desktop. It's absolutely essential that you test your controls before distributing them; the desktop is the most commonly used testing ground. Obviously, you'll also want to test ATL controls on a Web page to verify that they work in that environment as well. In fact, the Internet is one of the places where ATL-based controls really excel because of their small size and nondependence on external resources such as the MFC library.

Creating Simple ActiveX and COM+ Controls

It's finally time for the main event—we're going to look at some example components created using different methodologies. The following sections show you how to create the same ActiveX control using three different methods. Using this technique will allow you to see how the various methods compare. In other words, you'll see that it takes longer to develop an ATL version of the component than the MFC version, and that development time for the COM+ version falls in the middle.

The example component is a special kind of a pushbutton—in this case, a multiple state pushbutton. I chose a pushbutton because they'recommonly used in every programming environment. You'll even find them in odd places such as help (HLP) files. In fact, pushbuttons are the closest thing to a universal component that you can find. Another reason for choosing a pushbutton is ease of development. It's important not to get mired in a wealth of programming details when attempting to make a methodology comparison of this sort. With that in mind, our simple pushbutton will allow you to exercise a certain number of restricted methods, using a limited number of properties of various types. Table 13-1 describes the properties, methods, and events that each version of the pushbutton control implements.



In addition to the methods defined in Table 13-1, each property has both a get and set method as well. The get method retrieves the current property value, whereas the set method changes the current property value.

Table 13-1
Example Pushbutton Properties, Methods, and Events

Name	*Type*	*Description*
ModalResult	Property	Contains the current pushbutton result code. This code gets returned when the user requests the current pushbutton state.
ButtonType	Property	Determines what type of button is presented. The multistate feature of this pushbutton allows it to transition between several known states.
Caption	Property	The current string displayed on the button face that tells the user what function will be selected when the pushbutton is clicked.
Enabled	Property	Turns the pushbutton on or off. The user can't press the button when it's off.
Font	Property	Controls the typeface, size, and style of font used to display the caption.

Name	*Type*	*Description*
DoChange	Method	Sets the component's state to the next selection in the list without generating an OnClick event. This option allows you to do things such as set the beginning state of the button to a specific setting.
DoClick	Method	Imitates a click on the pushbutton, which results in an OnClick event and change in state to the next selection in the list.
SetUser	Method	Creates a set of user-defined states.
GetUser	Method	Returns the set of user-defined states.
OnClick	Event	Allows an application to detect when the user has clicked the pushbutton. Obviously, a full-fledged pushbutton probably implements other events so that you can detect things such as a key press instead of a click.

The MFC Version

The first method we'll use to create an ActiveX control relies on MFC and a fairly in-depth wizard. In fact, you'll find that this first technique requires the least amount of hand coding because a lot of the work is done for you. For example, as you define the various properties, methods, and events that you want the control to support, Visual C++ automatically creates the required method shells for you. The following sections show you how to create the example ActiveX control using the MFC technique.

Using the MFC method to create an ActiveX control requires a four-step process.

1. Create the application shell.
2. Use the MFC ClassWizard to add methods, properties, and events.
3. Design any required resources such as property pages and the About dialog box.
4. Add the code required to make the various control features work. (In most cases, you already have an empty method available.)

CD-ROM Location: Chapt12\MultPB1
Program Name: MultPB1.OCX
Source Code Modules in Text: MultPB1.CPP, MultPB1.H, StdAfx.H

Creating the MFC Version Program Shell

The first thing we need to do to create an MFC-based ActiveX control is create a program shell. What you end up with at the end of this section is an empty control with certain characteristics, but no resources or defined interface elements. The following procedure gets you started.

All of the examples in this chapter were created using the Enterprise Edition of Visual C++ 6.0. There are definite differences between this version and older versions of Visual C++. You may also note a small difference in the screen shots shown in this section and the ones that you see on your machine. What you see depends on what edition of Visual C++ that you use, the configuration of your setup, and which service packs you install (there was one service pack at the time of this writing).

Try This

1. Use the File | New command to display the New dialog box.
2. Highlight the MFC ActiveX ControlWizard icon, then type a name for the project in the Project Name field. The example uses a name of MultPB1.
3. Click OK. You can see the MFC ActiveX ControlWizard - Step 1 of 2 dialog box shown in Figure 13-1. This dialog box allows you to choose how many components appear in the file and general control features like the use of a runtime license. We'll use the default settings for this example, which includes a single component in the file, no runtime license, no help file, and source code comments.

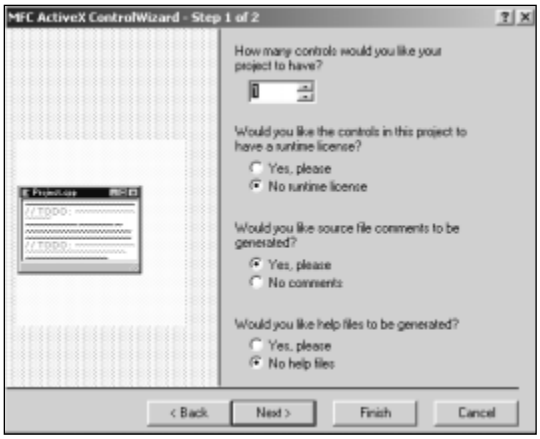

MFC ActiveX ControlWizard - Step 1 of 2
How many controls would you like your project to have?
1
Would you like the controls in this project to have a runtime license?
Yes, please
No runtime license
Would you like source file comments to be generated?
Yes, please
No comments
Would you like help files to be generated?
Yes, please
No help files
< Back
Next >
Finish
Cancel

Figure 13-1: The first MFC ActiveX ControlWizard dialog box allows you to configure the general component settings.

4. Click Next. You'll see the MFC ActiveX Control Wizard - Step 2 of 2 dialog box shown in Figure 13-2. This dialog box allows you to choose specific component settings. You choose the component that you want to work with using the drop-down listbox at the top of the dialog box.

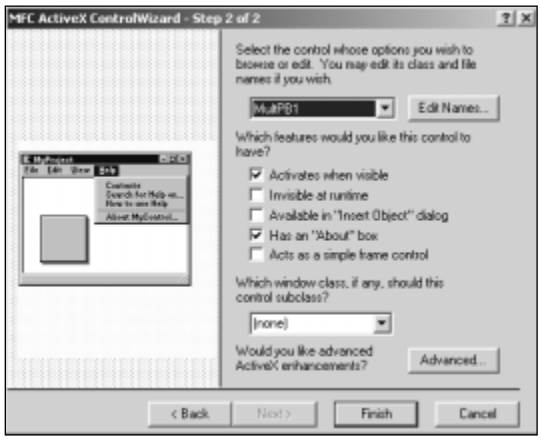

MFC ActiveX ControlWizard - Step 2 of 2
Select the control whose options you wish to browse or edit. You may edit its class and file names if you wish.
MultPB1
Edit Names...
Which features would you like this control to have?
Activates when visible
Invisible at runtime
Available in "Insert Object" dialog
Has an "About" box
Acts as a simple frame control
Which window class, if any, should this control subclass?
(none)
Would you like advanced ActiveX enhancements?
Advanced...
< Back
Next >
Finish
Cancel

Figure 13-2: The second MFC ActiveX ControlWizard dialog box allows you to configure specific settings for each component in the file.

5. Click Edit Name and you see the Edit Names dialog box shown in Figure 13-3. This dialog box allows you to change things like the names of the files used to hold the code for the component. Notice that there are entries for both the component and an associated property page. The MFC ActiveX ControlWizard creates a single property page for you by default. The entry of real interest on this dialog box is the Type Name field for the ActiveX control. This is the name you'll see when you insert the control into an application from a control list, so it pays to use something a little more descriptive than what appears in this case.

6. Type **Multiple State Pushbutton Control (MFC)** in the Type Name field, then click OK. You'll return to the MFC ActiveX ControlWizard - Step 2 of 2 dialog box.

7. Click Advanced. You can see the Advanced ActiveX Features dialog box shown in Figure 13-4. This dialog box allows you to add performance-enhancing features to your component. The problem is that not all of the features work in all environments, so you need to consider the environment as part of choosing a feature here. I talk more about this problem in the "ActiveX Gotchas for Visual C++ and Visual Basic Programmers" section of this chapter.

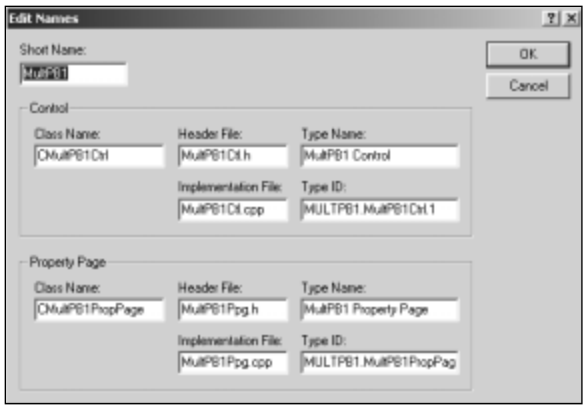


Figure 13-3: The Edit Name dialog box allows you to change the names of the class, files, and type information for a component and associated property page.

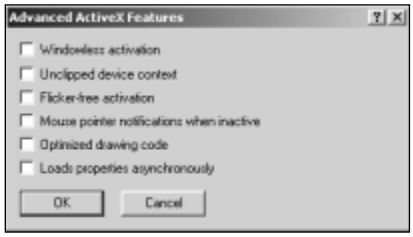


Figure 13-4: The Advanced ActiveX Features dialog box's main purpose is to enhance the performance of your ActiveX control in some way.

8. Check the Optimized drawing code option, then click OK to close the dialog box. In many cases, we also could have checked the Flicker free activation option as well, but that feature may have interfered with the Enabled property since the control won't draw itself the same in both the active and inactive states. It's important to consider things like this when choosing options on this dialog box. At this point, we need to choose the main component options.
9. Choose the BUTTON option from the Which window class, if any, should this control subclass field (shown as "subclass?"). Choosing this option means that our pushbutton automatically inherits some pushbutton features.
10. Click Finish. You see the New Project Information dialog box shown in Figure 13-5. Notice that this dialog box includes all of the feature changes we made, including those on ancillary dialogs — except for the Type Name field change on the Edit Name dialog box.
11. Click OK. Visual C++ creates the project shell for you.

New Project Information
MFC ActiveX ControlWizard will create a new skeleton project with the following specifications:
MFC ActiveX Control project MultPB1.mak
DLL name: MultPB1.ocx
DLL initialization code in MultPB1.h and MultPB1.cpp
Type library source in MultPB1.odl
MultPB1 Control:
Control class CMultPB1Ctrl in MultPB1Ctl.h and MultPB1Ctl.cpp
Property page class CMultPB1PropPage in MultPB1Ppg.h and MultPB1Ppg.cpp
Toolbar bitmap in MultPB1Ctl.bmp
Activates when visible
Uses optimized drawing
Has an "About" box
Subclasses a Windows BUTTON control
Project Directory:
D:\WIN_NT\CHAPT12\MultPB1
OK
Cancel

Figure 13-5: The New Project Information dialog box shows how the new pushbutton control shell is constructed.

Adding Methods, Properties, and Events to the MFC Version

At this point, you have a blank pushbutton control. If you compile this project and place the resulting control on a dialog box, you can press the button, and it responds by moving in and out. Of course, that's the limit of what the control can do. In this section, we'll add the methods, properties, and events that will allow the control to do more than it can now. The following procedure shows you how to add the methods and properties.

Try This

1. Use the View | ClassWizard command to display the MFC ClassWizard dialog box.
2. Choose the Automation tab and you see a dialog box like the one shown in Figure 13-6. This is where we add the methods and properties to our example.
3. Click Add Property and you see the Add Property dialog box shown in Figure 13-7. This dialog box allows you to add two different types of properties to your control: stock and custom. In addition, you can choose the implementation for each property — the way that it gets added to and managed by the control. The Caption, Font, and Enabled properties are all stock properties. A stock property is automatically managed for you by the control code. The two custom properties — ButtonType and ModalResult — require specialized code.

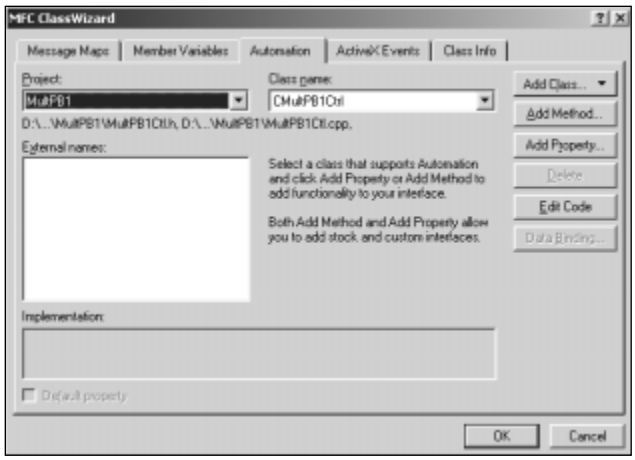


Figure 13-6: The Automation tab of the MFC ClassWizard dialog box is where you add methods and properties to your control.

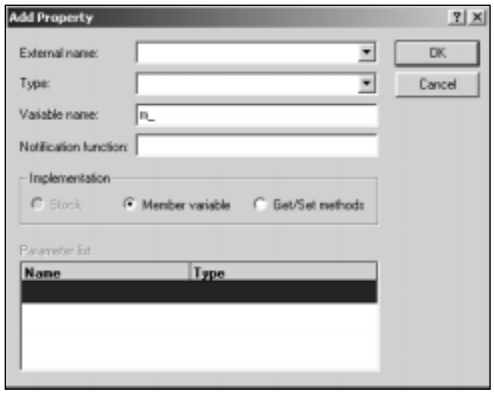


Figure 13-7: The Add Property dialog box allows you to add either stock or custom properties to your control.

4. Choose the Caption property from the External name field. Notice that Visual C++ automatically fills in the rest of the information required for this property for you. It also checks the Stock implementation radio button. Check the Get/Set methods implementation option if you want to create a custom implementation of this stock property.

5. Click OK. Visual C++ adds the new property to the Automation tab of the MFC ClassWizard dialog box for you.
6. Repeat Steps 3 through 5 for the Font and Enabled properties. Visual C++ adds new entries to the Automation tab for each stock property you select.
7. Repeat Steps 3 through 5 for the ButtonType and ModalResult properties. In this case, you need to choose some options on the Add Properties dialog box. Choose Short in the Type field and Get/Set methods in the implementation. We could use a Member variable implementation, but you'll find that the Get/Set methods implementation works better when using the control with other languages. If you develop a control for exclusive use by Visual C++, you'll find that the Member variable option actually is less work from a programming perspective. At this point, we can add the four methods for this control: DoChange( ), DoClick( ) GetUser( ), and SetUser( ).
8. Click Add Method and you see the Add Method dialog box shown in Figure 13-8. As with the Add Property dialog box, this dialog box allows you to add both stock and custom methods to the control.

Figure 13-8: The Add Method dialog box allows you to add both stock and custom properties to a control.

9. Add the stock DoClick( ) method by selecting DoClick from the External name field and clicking OK. Visual C++ displays the new method on the Automation tab of the MFC ClassWizard dialog box. Now, we can add the three custom methods.
10. Click Add Method to display the Add Method dialog box.
11. Type **DoChange** in the External name field. Visual C++ automatically uses the same name for the Internal name field.
12. Select Void from the Return type field list; then click OK.

13. Repeat Steps 10 through 12 for the GetUser( ) and SetUser( ) methods. However, the Return type for GetUser( ) is BSTR, and the Return type for SetUser( ) is BOOL. You also need to add a parameter to the SetUser( ) method. Just type the name of the parameter (**lpszUserString**) in the Name column and choose the parameter type in the Type column as shown in Figure 13-9.

Figure 13-9: Adding parameters to either methods or properties is a relatively simple matter.

14. Verify that you added all of the properties and methods. The Automation tab of the MFC ClassWizard should look similar to the one shown in Figure 13-10. Notice that Visual C++ marks stock properties with an S, custom properties with a C, and all methods with an M.

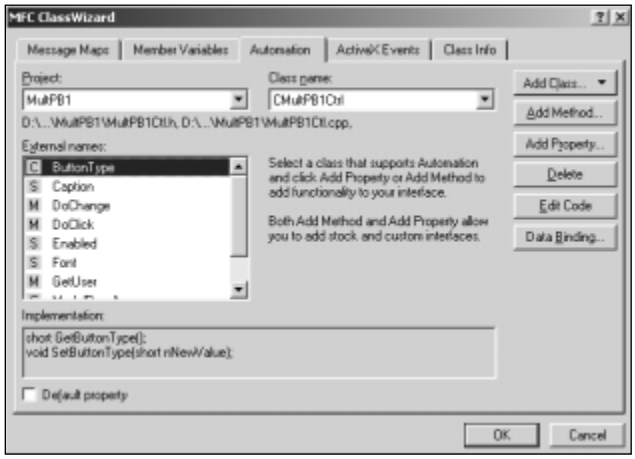


Figure 13-10: Visual C++ makes it easy for you to see what kinds of methods and properties you add to the control.

15. Click OK to close the MFC ClassWizard to ensure that all of the method and property settings get recorded properly.

Registering the various properties we need with the MFC ClassWizard may create the functions required to manipulate them, but we still don't have a way to keep track of the values they hold. What we need is a variable that will hold the value of what the set method for a property receives and the get method provides to the client. In addition, we need a variable to work with the property page when we create it. For this reason, you should create member variables for each of the custom properties that we just created (ButtonType and ModalResult). Adding the required variables is easy. The following procedure shows you how.

Try This

1. Right-click the CmultPB1Ctrl folder in the ClassView tab of the Workspace window; then choose Add Member Variable from the context menu. You should see the Add Member Variable dialog box shown in Figure 13-11.

Figure 13-11: The Add Member Variable dialog box allows you to add holding variables for the custom properties you create.

2. Type **short** in the Variable Type field. Always use the same type for the member variable that Visual C++ returns for the property.

Normally, Visual C++ returns the same type of variable as the property. For example, if you have a short integer property, then you need to make the member variable a short integer. However, there are some exceptions to the rule. The most important exception is BSTRs. Visual C++ normally converts these property values to a LPCTSTR for you. You need to keep the potential differences between the property variable type and the variable type that Visual C++ returns in mind when you create the member variable.

3. Type **m_buttonType** in the Variable Name field.
4. Click OK. Visual C++ adds the member variable to the control for you.
5. Repeat Steps 1 through 5 for the ModalResult property. Be sure to use m_modalResult for the member variable name.

At this point, you have all of the methods and properties for the example control in place so it's time to add the one event that we'll work with. The following procedure shows you how.

Try This

1. Use the View | ClassWizard command to display the MFC ClassWizard dialog box, then choose the ActiveX Events tab.
2. Click Add Event. You should see the Add Event dialog box shown in Figure 13-12. This is where you add either stock or custom events to your control. Notice that this dialog box looks similar to the ones that we used to add properties and methods.

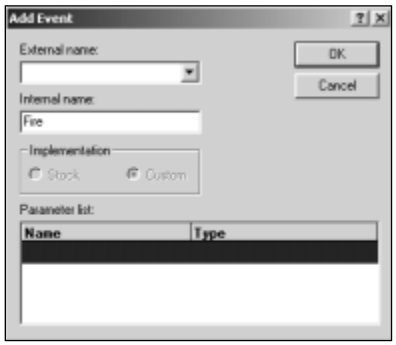


Figure 13-12: The Add Event dialog box allows you to add custom or stock events to your control.

3. Choose Click from the External name field list. As usual, Visual C++ fills in the rest of the required fields for you.
4. Click OK to add the event. The ActiveX Events tab of the MFC ClassWizard dialog box should look similar to the one shown in Figure 13-13.

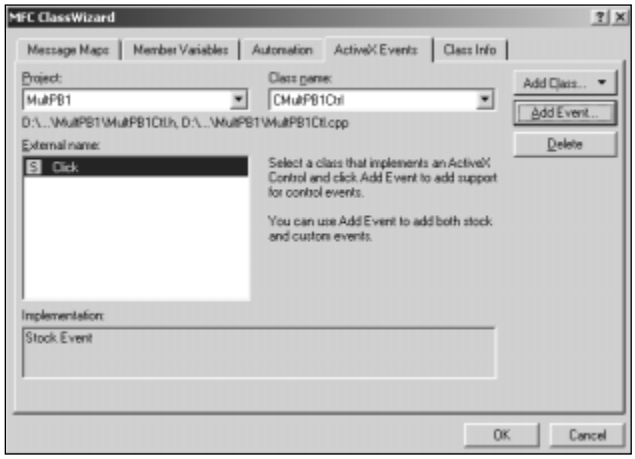


Figure 13-13: The ActiveX Events tab displays events by type: C for custom and S for stock.

Designing Resources for the MFC Version

It may not seem at first like you'd require any resources at all for an ActiveX control unless it displayed informational dialog boxes, worked with a graphic element, or performed some other type of user interface. However, there are four resources that you can count on creating for even a minimal ActiveX control. They include:

- **About Dialog Box:** Works the same as an About dialog box for an application by displaying useful vendor and control information, along with supplementary information when needed.
- **Control Bitmap:** Used to display the control in a toolbox or control list. The bitmap you use needs to show the control's functionality so that the programmer can pick it out of a group of icons relatively fast.
- **Version Information:** Tells the user and programmer alike what version of a control is being used. This information can be used a wide variety of purposes, including technical support when a control causes problems in a commercial application.
- **Properties Dialog Box:** Allows the programmer to set the various properties provided by your control. Some environments don't provide a Property Window that automatically displays a list of properties for the control, so the Properties dialog box needs to provide a complete list of properties associated with the control.

Let's begin with the About dialog box. The About dialog box actually consists of two resources, the dialog itself (IDD_ABOUTBOX_MULTPB1) and an associated icon (IDI_ABOUTDLL). The icon defaults to the same MFC icon that's used for applications (IDR_MAINFRAME). This icon comes in two sizes, though you can always add more to the list. The "Icons" section of Chapter 4 discusses the topic of application icons in detail, so I won't go into detail here. Figure 13-14 shows the 32 X 32 -pixel icon we'll use for this example. The 16 X 16-pixel icon is the same, just smaller.

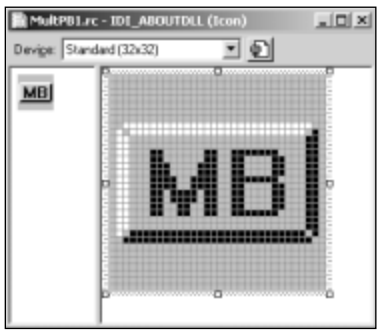


Figure 13-14: The IDI_ABOUT icon is displayed in the About dialog box.

The About dialog box itself is fairly complete once you add a custom icon. However, it's usually a good idea to include contact information so programmers can figure out where to get additional information quickly. Figure 13-15 shows a simple, but effective, About dialog box for an ActiveX control. Note that this dialog box contains four essential pieces of information: control name, version number, company name, and contact information.

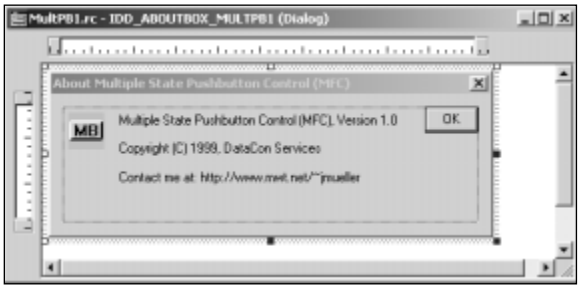

MultPB1.rc - IDD_ABOUTBOX_MULTPB1 (Dialog)
About Multiple State Pushbutton Control (MFC)
MB
Multiple State Pushbutton Control (MFC), Version 1.0
OK
Copyright (C) 1999, DataCon Services
Contact me at: http://www.mvet.net/~jmueller

Figure 13-15: A well-designed About dialog box tells the programmer enough about the control to get help using it.

The control bitmap (IDB_MULTPB1) gets displayed whenever the programmer places the OCX on the toolbar. It also gets displayed in a variety of other places such as component lists and within utilities that work with components. Figure 13-13 shows what this bitmap should look like, although you'll probably need to resize the art to fit within the current bitmap.

Adding version information to your control using the VS_VERSION_INFO resource is important for several reasons, not the least of which is ensuring that both programmers and users alike can determine which version of your control they're using. Often an old version of a control is the culprit when an application fails to work as expected. For example, you may have fixed a bug in a newer version of the control. In addition, there are environmental changes that you need to take into account. Not only do controls need to keep pace with changing application needs, but also they need to change to take advantage of new operating system features. Needless to say, keeping the version information up to date is vitally important. Figure 13-16 shows the version information for this example; obviously your control's version information slightly differs.

For a more complete discussion of how the version information resource works, read the "String and Version Tables" section of Chapter 4.

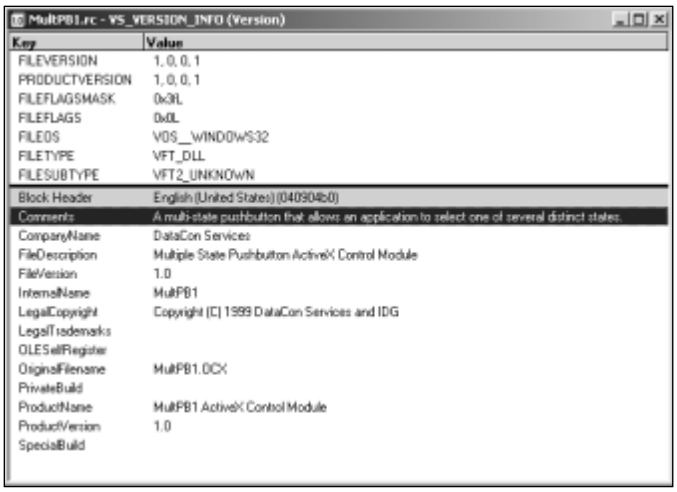


Figure 13-16: The VS_VERSION_INFO resource contains the version-specific information for your control.

The final resource that we need to modify is the property page. The property page requires a little more work than the other resources that we've worked with so far because it has to interact with the properties that we've created. You'll also spend more time designing the property page and determining if you need more than one to get the job done. Using more than one property page allows you to keep the properties organized by type, but it also requires more work. In addition, make sure you actually have enough properties to warrant more than one property page.

There are two standard sizes of a property page supported by Visual C++ and most other Visual Studio products. The small size, which is the default for an OCX, is 250 (62 pixels. The large size is 250 (110 pixels, which is usually large enough to hold the property controls for a simple ActiveX control. Make sure you use one size or the other when creating an Active control. Nothing bad will happen if you don't use a standard size, but users get warning messages saying that you didn't use a standard-sized property page. Some users may interpret this as a problem with the ActiveX control, when in reality it doesn't change the functionality of the control at all.

The first thing we need to do is determine the size of the property page. A 250 (62 pixels property page is normally large enough to hold five or six controls without too much crowding; in some cases, you might be able to go as high as 10 controls, but this usually does look crowded. The 250 (62 size is too small for our purposes, so we need to resize it to the large property page size of 250 (110.

Adding some controls to the property page comes next. This control supports three default multiple button settings: On/Off, Yes/No, and Red/Yellow/Green. When the user clicks the button for a specific setting, the button toggles between the states that you set for it. We'll also support a User Mode, which defaults to a standard button if you don't provide a user string for it. As a result, we need four radio buttons on the left side of the property page to control what type of button gets created.

The ModalResult property is actually connected to the kind of button being used at the time, so we don't need to create a property page entry for it. We could, however, add a read-only output so that the user knows what ModalResult value was used for the current button setting.

The Font property is controlled by its own special property page that we add later in the example.

That leaves the Caption and Enabled properties, which we'll add on the right side of the property page. The Enabled property uses a checkbox control. The Caption property gets used only if we create a standard button, but it's an important property to add using a standard edit box control. Figure 13-17 shows the layout for the property page. Table 13-2 contains the property settings you need for each of the controls on the property page.

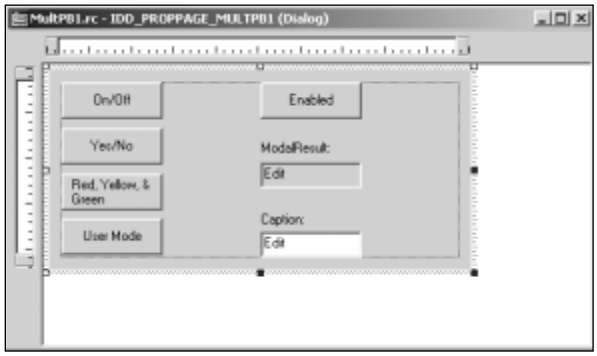


Figure 13-17: The property page will allow us to change most of the properties for the MultiPB1 ActiveX control.

Table 13-2
MultiPB1 Property Page Control Settings

Control ID	*Property*	*Value*
IDC_YES_NO	Height	20
	Width	60
	Push-like	Checked
	Group	Checked
	Tab stop	Checked
	Caption	Yes/No
IDC_ON_OFF	Height	20
	Width	60
	Push-like	Checked
	Tab stop	Checked
	Caption	On/Off
IDC_RED_YELLOW_GREEN	Height	20
	Width	60
	Push-like	Checked
	Multiline	Checked
	Tab stop	Checked
	Caption	Red, Yellow, and Green
IDC_USER_MODE	Height	20
	Width	60
	Push-like	Checked
	Tab stop	Checked
	Caption	User Mode
IDC_ENABLED	Height	20
	Width	60
	Push-like	Checked
	Caption	Enabled
IDC_MODALRESULT	Read-only	Checked
	Width	60
IDC_CAPTION	Width	60

Now that we have a property page, we need to connect the buttons and edit boxes on this page with the properties they're supposed to represent. The following steps show you how.

Try This

1. Ctrl+Double-click Yes/No. You should see the Add Member Variable dialog box shown in Figure 13-18. This dialog box connects controls to the rest of an application in a variety of ways. Even though you use the same dialog box to perform each task, the kinds of entries you make determine how the control gets connected.

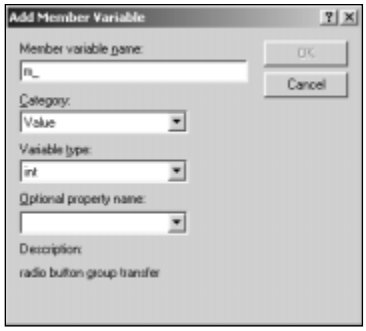


Figure 13-18: The Add Member Variable dialog box allows you to create a connection between a control and a property.

2. Type **m_buttonType** in the Member variable name field. This field creates a member variable that you can access within your application code when working with the property page.
3. Type **ButtonType** in the Optional property name field. The contents of this field link the control to the property value of your ActiveX control, so it's important to type a property name that actually exists. The drop-down listbox contains a list of the most common property values so that you can choose a common property without typing.
4. Click OK. Visual C++ creates the member variable for you.
5. Repeat Steps 1 through 4 for the Enabled, Modal Result, and Caption controls. You need to substitute appropriate field entries for each of these controls. Table 13-3 shows what values to use.

Writing the MFC Version Code

Gtvu

We're finally ready to add some code. You'll find that all of the preparation we've done was well worth the effort because the coding requirements are pretty minimal. The following sections show you how to write each section of code.

Table 13-3
Add Member Variable Dialog Entry Values

Control	*Field*	*Value*
Enabled	Member variable name	m_enabled
	Optional property name	Enabled
Modal Result	Member variable name	m_modalResult
	Optional property name	ModalResult
Caption	Member variable name	m_caption
	Optional property name	Caption

Creating Persistent Properties

The first thing we have to do is make the custom properties for the control persistent. (The stock properties are persistent when you create them.) Listing 13-1 shows the code you need to add to the DoPropExchange( ) method in bold.

Listing 13-1: **Making the Properties Persistent Source Code**

```
void CMultPB1Ctrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
    // Make our custom properties persistent.
    PX_Short(pPX, "ModalResult", m_modalResult, 1);
    PX_Short(pPX, "ButtonType", m_buttonType, 0);

}
```

As you can see, making a property persistent requires a single call. There are a variety of PX_ functions that you can use for the various variable types. Listing 13-1 shows the PX_Short( ) function because our property values are of the short type. You need to provide at least three arguments for the PX_ functions. The first argument contains a pointer to the CPropExchange variable pointer. That value is supplied with the DoPropExchange( ) method. You also need to provide the text name of your property and the member variable used to track its contents. Finally, you can supply a default value for the property, but this isn't a requirement.

Defining the Get/Set Methods for the Properties

The get/set methods for our properties come next. However, before we can create the get/set methods, there are three member variables that we need to create. The first variable (named aDisplayValues) is a CArray<CString, CString> used to hold the values that display each time the user clicks the control or the programmer issues a DoChange( ) command. The second (named oUserString) is a CString variable used to hold the user string. If the user string is blank, then the button behaves as a normal button. On the other hand, if the user string contains a valid array of button values, then the control uses those values instead of the preprogrammed alternatives. The third member variable, iCurrentChoice, is an int that holds the current button choice from the array of choices in aDisplayValues. All three of these member variables appear in the CMultPB1Ctrl class, just like our other member variables. I discussed how to create member variables in the "Designing Resources" section, so I won't discuss that procedure here again.

The CArray<CString, CString> is a template class. The first argument tells Visual C++ that you want to store CString objects in the CArray. The second argument tells Visual C++ that you want to use CString objects to access the array element. A template class requires some special support. You need to open the StdAfx.H file and add the following lines of code to it:

```
// Include support for template classes.
#include <afxtempl.h>
```

Now that we have all of the required support in place, let's look at the source for working with the two custom properties: ModalResult and ButtonType. Listing 13-2 shows the code you need to add for the four get/set methods.

Listing 13-2: ModalResult and ButtonType Get/Set Methods Source Code

```
short CMultPB1Ctrl::GetButtonType()
{
    // Return the current ButtonType value.
    return m_buttonType;
}

void CMultPB1Ctrl::SetButtonType(short nNewValue)
{
    // Set the ButtonType value.
    m_buttonType = nNewValue;

    // Determine which ButtonType has been
    // selected, then set the dependent properties
    // and member variables to match.
    switch (nNewValue)
```

```
{
case 0:
    COleControl::SetText("On");
    m_modalResult = 1;
    break;
case 1:
    COleControl::SetText("Yes");
    m_modalResult = 3;
    break;
case 2:
    COleControl::SetText("Red");
    m_modalResult = 5;
    break;
case 3:
    COleControl::SetText("Button");
    m_modalResult = -1;
    break;
}

// Set the currently selected multi-state
// option value.
iCurrentChoice = 0;

// Clear the current display array contents.
aDisplayValues.RemoveAll();
// Determine which button the user has selected, then
// fill the display array with an appropriate set of
// display values.
switch (m_buttonType)
{
case 0:
    aDisplayValues.Add(CString("On"));
    aDisplayValues.Add(CString("Off"));
    break;
case 1:
    aDisplayValues.Add(CString("Yes"));
    aDisplayValues.Add(CString("No"));
    break;
case 2:
    aDisplayValues.Add(CString("Red"));
    aDisplayValues.Add(CString("Yellow"));
    aDisplayValues.Add(CString("Green"));
    break;
 case 3:
    if (oUserString.GetLength() > 0)
        SetUser(oUserString.GetBuffer(80));
}

SetModifiedFlag();
```

Continued

Listing 13-2: *(continued)*

```
}

short CMultPB1Ctrl::GetModalResult()
{
    // Return the current ModalResult value.
    return m_modalResult;
}

void CMultPB1Ctrl::SetModalResult(short nNewValue)
{
    // Set the ModalResult value.
    m_modalResult = nNewValue;

    // Determine which ModalResult has been
    // selected, then set the dependent properties
    // and member variables to match.
    switch (nNewValue)
    {
    case 1:
        COleControl::SetText("On");
        iCurrentChoice = 0;
        m_buttonType = 0;
        break;
    case 2:
        COleControl::SetText("Off");
        iCurrentChoice = 1;
        m_buttonType = 0;
        break;
    case 3:
        COleControl::SetText("Yes");
        iCurrentChoice = 0;
        m_buttonType = 1;
        break;
    case 4:
        COleControl::SetText("No");
        iCurrentChoice = 1;
        m_buttonType = 1;
        break;
    case 5:
        COleControl::SetText("Red");
        iCurrentChoice = 0;
        m_buttonType = 2;
        break;
    case 6:
        COleControl::SetText("Yellow");
        iCurrentChoice = 1;
        m_buttonType = 2;
        break;
```

```
    case 7:
        COleControl::SetText("Green");
        iCurrentChoice = 2;
        m_buttonType = 2;
        break;
    case -1:
        COleControl::SetText("Button");
        iCurrentChoice = 0;
        m_buttonType = 3;
        break;
    }

    // Clear the current display array contents.
    aDisplayValues.RemoveAll();
    // Determine which button the user has selected, then
    // fill the display array with an appropriate set of
    // display values.
    switch (m_buttonType)
    {
    case 0:
        aDisplayValues.Add(CString("On"));
        aDisplayValues.Add(CString("Off"));
        break;
    case 1:
        aDisplayValues.Add(CString("Yes"));
        aDisplayValues.Add(CString("No"));
        break;
    case 2:
        aDisplayValues.Add(CString("Red"));
        aDisplayValues.Add(CString("Yellow"));
        aDisplayValues.Add(CString("Green"));
        break;
    case 3:
        if (oUserString.GetLength() > 0)
            SetUser(oUserString.GetBuffer(80));
    }

    SetModifiedFlag();
}
```

Let's talk about the two simple methods first. The two Get methods are essentially the same. All we do is return the value of the internal member variable to the requestor.

The Set methods also have some similarities. The order in which the two Set methods perform the setup differ because the way the control has to react to property changes for each property differs. Both go through the same four-step process of setting up the various control properties.

Here are the four steps that the code follows to set the ButtonType or ModalResult properties:

1. Change the button caption.
2. Set the ModalResult and ButtonType values.
3. Set the current button selection using iCurrentChoice.
4. Clear, then fill the DisplayValues array with the set for values for button selections. Make sure that the variables you use to fill the array actually match the template settings. In this case, I made sure that I used the CString constructor to fill the array.

Notice that the built-in properties always follow the same setting steps in both cases, but that the user mode string, oUserMode, setup differs. That's because the user mode has to be parsed and the control set up once the individual display values passed by the user are known. Rather than write the parsing routines twice, I merely call the SetUser( ) method when appropriate to perform all of the required work. It's also important to note that you need to pass SetUser an LPCTSTR. That's easy to do using the GetBuffer( ) method of the CString class. Make sure that you create a large enough buffer to hold the entire string contained in oUserMode. One way to ensure this is to use the CString class GetLength( ) method. (This example leaves out much of the error trapping you normally perform in a control for the sake of simplicity.)

Adding Code for the Control's Methods

At this point, we can create the three methods for our control: DoChange( ), GetUser( ), and SetUser( ). Remember that DoChange( ) allows you to change the current button state without firing an OnClick( ) event. GetUser( ) and SetUser( ) control the user mode string used to create custom button sequences. Listing 13-3 shows the code you need to add to the example to create these three methods.

Listing 13-3: DoChange(), GetUser(), and SetUser() Methods Source Code

```
void CMultPB1Ctrl::DoChange()
{
    // Determine what the next button position is.
    if (iCurrentChoice < aDisplayValues.GetSize() - 1)
    {
        iCurrentChoice++;
        m_modalResult++;
    }
    else
    {
```

```
        m_modalResult = m_modalResult - iCurrentChoice;
        iCurrentChoice = 0;
    }

    // Display the button text for the current position.
    if (aDisplayValues.GetSize() > 0)
        COleControl::SetText(aDisplayValues.GetAt(iCurrentChoice));

}

BSTR CMultPB1Ctrl::GetUser()
{
    // Return the user string.
    return oUserString.AllocSysString();
}

BOOL CMultPB1Ctrl::SetUser(LPCTSTR lpszUserString)
{
    int     iCounter;       // A counter variable.
    CString oSingleEntry;   // A single entry.
    char    cTemp;          // Temporary character storage.

    // Set the user string to a new value.
    oUserString = lpszUserString;

    // Make sure we have something to display.
    if (oUserString.GetLength() == 0)
    {
        // If not, then set all of the require user mode
        // values back to their default state.
        COleControl::SetText("Button");
        m_modalResult = -1;
        aDisplayValues.RemoveAll();
        iCurrentChoice = 0;

        // Return without parsing the string.
        return TRUE;
    }

    // Clear the display value array.
    aDisplayValues.RemoveAll();

    // Parse the string and create the display array.
    for (iCounter = 0; iCounter < oUserString.GetLength() - 1; iCounter++)
    {
        cTemp = oUserString.GetAt(iCounter);
        if (strncmp(&cTemp, ";", 1) == 0)
        {
            // Add a new entry to the display array.
```

Continued

Listing 13-3: *(continued)*

```
            aDisplayValues.Add(oSingleEntry);
            // Reset the single entry variable.
            oSingleEntry = "";
            // Set the caption to the first user mode string value.
            COleControl::SetText(aDisplayValues.GetAt(0));
            // Set ModalResult to 1 so the client can process the results.
            m_modalResult = 1;
        }
        else
            // Place the next character in the single entry variable.
            oSingleEntry += oUserString.GetAt(iCounter);
    }

    return TRUE;
}
```

The DoChange( ) method code can't rely on a specific number of aDisplayValues array entries, so the code avoids assuming anything about the number entries that it might find. We do know, however, that iCurrentChoice and m_modalResult are locked together when tracking the current string displayed on the button. In other words, every time the user clicks the button, the user either moves to the next choice, or, if there aren't any more choices, moves back to the first choice. The if statement at the beginning of the method comes from here. Once the control knows what position within the aDisplayValues array that it's supposed to display, it has to ensure that there actually is a value to display. Remember that a default user mode button doesn't have any aDisplayValues entries because it works as a standard button. If there's a value to display, the iCurrentValue variable is used to choose a specific entry within the array and apply it to the button caption.

The GetUser( ) method is a little more complicated than the Get methods used by the properties for our control. Language products such as Visual Basic rely on the BSTR rather than the Visual C++ equivalents. The AllocSysString( ) method of the CString class allows you to create a BSTR from a CString with relative ease.

The final method, SetUser( ), has to perform quite a bit of work to get the string passed by the user to react as a series of button selections. However, before you can understand the code, you have to understand how the user mode string is formatted. I chose a relatively easy formatting method. A semicolon separates each string; the user needs to supply two semicolons at the end of the string. So a two-entry string containing One and Two as button values looks like this:

```
One;Two;;
```

The first thing that the code does is place the value passed by the client into oUserString. Once this is accomplished, the control verifies that there's a value in oUserString to process. If not, then the code sets all of the user mode settings to default, then exits. This arrangement allows a user to pass a blank string to clear the button for normal operation.

Once the SetUser( ) method determines that there's a string to process, it clears the current values in the aDisplayValues array. The control then enters a loop that looks at each character in the oUserString variable. If the character is a semicolon, it signals the end of a button entry; the value in the single entry variable, oSingleEntry, is placed in aDisplayValues then cleared. In addition, the first entry in aDisplayValues is displayed on the button face, and m_modalResult is set to 1, which allows the control to output ModalResult values for the control. If the character is anything else, it gets added to oSingleEntry and the loop continues to the next character.

Defining the OnClick( ) Event

The only basic function we have left is the OnClick( ) event. This event determines what happens when the user clicks the button. The source code you need to make the OnClick( ) event functional appears in Listing 13-4.

Listing 13-4: **OnClick( ) Event Method Source Code**

```
void CMultPB1Ctrl::OnClick(USHORT iButton)
{
     // Determine what the next button position is.
    if (iCurrentChoice < aDisplayValues.GetSize() - 1)
    {
        iCurrentChoice++;
        m_modalResult++;
    }
    else
    {
        m_modalResult = m_modalResult - iCurrentChoice;
        iCurrentChoice = 0;
    }

    // Display the button text for the current position.
    if (aDisplayValues.GetSize() > 0)
        COleControl::SetText(aDisplayValues.GetAt(iCurrentChoice));
    // Perform the default action.
     COleControl::OnClick(iButton);
}
```

This code should look pretty familiar. It's the same code that we used for the DoChange( ) method. However, there's one small difference. The OnClick( ) event also needs to fire an OnClick event for the client. This feature and any housekeeping chores are taken care of by a call to the OnClick( ) superclass.

Adding Some Initialization Code

Now that we have all of the basic pieces in place, there's one other task we have to perform to make this button functional. Even with all of the variables we've created and the persistence we've added to the properties, there are still two initialization tasks we have to perform to make the button react as anticipated. First, we need to ensure that the aDisplayValues array contains some values to work when the control is instantiated. Second, we need to ensure that iCurrentChoice is set consistently with the current ModalResult property value so that the OnClick( ) and DoChange( ) methods work properly. The best place to add this initialization code is to the PreCreateWindow( ) method, which already exists at the time you create the control. Listing 13-5 shows you this initialization code in bold.

Listing 13-5: **PreCreateWindow( ) Method Initialization Source Code**

```
BOOL CMultPB1Ctrl::PreCreateWindow(CREATESTRUCT& cs)
{
    // Define the control's base class.
     cs.lpszClass = _T("BUTTON");
    // Determine which button the user has selected, then
    // fill the display array with an appropriate set of
    // display values.
    switch (m_buttonType)
    {
    case 0:
        aDisplayValues.Add(CString("On"));
        aDisplayValues.Add(CString("Off"));
        break;
    case 1:
        aDisplayValues.Add(CString("Yes"));
        aDisplayValues.Add(CString("No"));
        break;
    case 2:
        aDisplayValues.Add(CString("Red"));
        aDisplayValues.Add(CString("Yellow"));
        aDisplayValues.Add(CString("Green"));
        break;
    }

    // Set the button up when the ModalResult setting was used
    // in place of ButtonType to configure the button.
```

```
    switch (m_modalResult)
    {
    case 2:
        iCurrentChoice = 1;
        break;
    case 4:
        iCurrentChoice = 1;
        break;
    case 6:
        iCurrentChoice = 1;
        break;
    case 7:
        iCurrentChoice = 2;
        break;
    }

    // Perform the default action.
     return COleControl::PreCreateWindow(cs);
}
```

Adding the Font Property Page

There are also some fit-and-finish items we need to address. The custom property page that we created already is part of the control, so you don't need to worry about whether the user will or will not see it. However, remember that we added the stock Font property to the control. This property will allow the user to change the font used to display the button's caption. Visual C++ provides some stock property pages that you can use to implement things like colors and fonts. Listing 13-6 shows the code you need to add in bold to the property page macros at the beginning of the MultPB1Ctrl.CPP file.

Listing 13-6: **Adding a Stock Font Property Page**

```
BEGIN_PROPPAGEIDS(CMultPB1Ctrl, 2)
     PROPPAGEID(CMultPB1PropPage::guid)
     PROPPAGEID(CLSID_CFontPropPage)
END_PROPPAGEIDS(CMultPB1Ctrl)
```

As you can see, there's nothing complicated about this process. When the user displays the property pages for this control, he or she sees both the custom and a stock Font property page (along with any pages provided natively by the language product). Notice that you also need to update the number of property pages in the BEGIN_PROPPAGEIDS( ) macro. Make absolutely certain you do this since Visual C++ may compile the code without error even if you don't.

Designing Listboxes for the ModalResult and ButtonType Properties

There's one last item we should add, although it's not necessary. The control is perfectly functional right now and you can use it without any trouble. However, the ModalResult and ButtonType properties won't be very easy to figure out for the casual programmer. Wouldn't it be nice if they both had drop-down listboxes that contained the acceptable values for the properties? You can add this feature with relative ease, but it requires you to manually add some support methods. The following procedure helps you add the three methods required to support drop-down listboxes for the ModalResult and ButtonType properties.

Try This

1. Use the View | ClassWizard command to display the MFC ClassWizard dialog.
2. Choose the Message Maps tab. This is where you can add handlers for messages that your control receives from either Windows or the client. If you don't add a specific message handler, then there's normally a default handler that takes care of the message for you. However, the vast majority of these default handlers don't do anything at all.
3. Choose CMultPB1Ctrl in both the Class name and Object IDs fields. These two fields determine where in the source code the message handler appears and what object it affects. You could, for example, add message handlers to controls on a dialog box using the Message Maps tab by selecting the proper value in the Object IDs field.
4. Highlight the OnGetDisplayString entry in the Messages field, then click Add Function. This message provides a feedback value to a client if the client requests a string equivalent for a property value. For example, even though the actual value of the ButtonType property might be 0, the property field in the client could say "0 - On/Off" instead, which is definitely easier to understand.
5. Highlight the OnGetPredefinedStrings entry in the Messages field, then click Add Function. This message tells the client what values are acceptable for a particular property. In addition, it provides a cookie that the control uses to actually change the property value.
6. Highlight the OnGetPredefinedValue entry in the Messages field, then click Add Function. At this point, the Message Maps tab of the MFC ClassWizard dialog box should look similar to the one shown in Figure 13-19. This message handler fills an array with the actual values used by the control to change the property. It receives the cookie that the OnGetPredefinedStrings handler created and translates this cookie value into something that the control can use.
7. Click OK. We're ready to add some code to these three new functions.

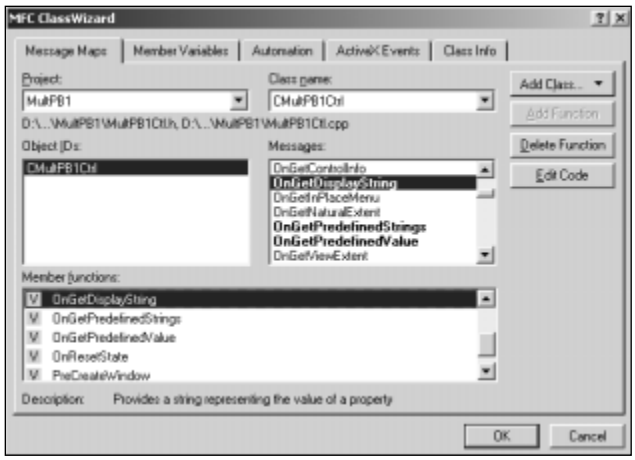


Figure 13-19: The Message Maps tab of the MFC ClassWizard dialog box shows the three methods we need to add to support listboxes for properties.

Visual C++ doesn't actually require all three of these methods to display the drop-down listboxes in the Property Window. However, other language products, such as Visual Basic, do require all three functions, so it's a good idea to add them. I talk more about the needs of products like Visual Basic in the "ActiveX Gotchas for Visual C++ and Visual Basic Programmers" section of this chapter. Listing 13-7 shows the code you need to make these three functions work.

Listing 13-7: OnGetDisplayString(), OnGetPredefinedStrings(), and OnGetPredefinedValue() Methods Source Code

```
BOOL CMultPB1Ctrl::OnGetPredefinedStrings(DISPID dispid, CStringArray*
pStringArray, CDWordArray* pCookieArray)
{
  // Determine which property to react to.
  switch (dispid)
  {

  // Set the values for the ButtonType property.
  case dispidButtonType:
    // Set the string values.
    pStringArray->SetSize(4, -1);
    pStringArray->SetAt(0, LPCTSTR("0 - On/Off"));
    pStringArray->SetAt(1, LPCTSTR("1 - Yes/No"));
```

Continued

Listing 13-7: *(continued)*

```
    pStringArray->SetAt(2, LPCTSTR("2 - Red/Yellow/Green"));
    pStringArray->SetAt(3, LPCTSTR("3 - User Mode"));

    // Set the cookie values.
    pCookieArray->SetSize(4, -1);
    pCookieArray->SetAt(0, 0);
    pCookieArray->SetAt(1, 1);
    pCookieArray->SetAt(2, 2);
    pCookieArray->SetAt(3, 3);

    // Return true to indicate that we've set the values.
    return true;
  // Set the values for the ModalResult property.
  case dispidModalResult:
    // Set the string values.
    pStringArray->SetSize(8, -1);
    pStringArray->SetAt(0, LPCTSTR("-1 - User Mode"));
    pStringArray->SetAt(1, LPCTSTR("1 - On"));
    pStringArray->SetAt(2, LPCTSTR("2 - Off"));
    pStringArray->SetAt(3, LPCTSTR("3 - Yes"));
    pStringArray->SetAt(4, LPCTSTR("4 - No"));
    pStringArray->SetAt(5, LPCTSTR("5 - Red"));
    pStringArray->SetAt(6, LPCTSTR("6 - Yellow"));
    pStringArray->SetAt(7, LPCTSTR("7 - Green"));

    // Set the cookie values.
    pCookieArray->SetSize(8, -1);
    pCookieArray->SetAt(0, -1);
    pCookieArray->SetAt(1, 1);
    pCookieArray->SetAt(2, 2);
    pCookieArray->SetAt(3, 3);
    pCookieArray->SetAt(4, 4);
    pCookieArray->SetAt(5, 5);
    pCookieArray->SetAt(6, 6);
    pCookieArray->SetAt(7, 7);

    // Return true to indicate that we've set the values.
    return true;
  }

  return COleControl::OnGetPredefinedStrings(dispid, pStringArray,
pCookieArray);
}

BOOL CMultPB1Ctrl::OnGetPredefinedValue(DISPID dispid, DWORD dwCookie, VARIANT*
lpvarOut)
{
  // Determine which property to react to.
```

```
  switch (dispid)
  {

  // Return a value for the predefined ButtonType string.
  case dispidButtonType:
    // Initialize the variant structure.
    VariantInit(lpvarOut);

    // Add the type and value of the output.
    lpvarOut->vt = VT_I2;
    lpvarOut->iVal = short(dwCookie);

    // Return true to indicate we've set the values.
    return true;
  // Return a value for the predefined ButtonType string.
  case dispidModalResult:
    // Initialize the variant structure.
    VariantInit(lpvarOut);

    // Add the type and value of the output.
    lpvarOut->vt = VT_I2;
    lpvarOut->iVal = short(dwCookie);

    // Return true to indicate we've set the values.
    return true;
  }

  return COleControl::OnGetPredefinedValue(dispid, dwCookie, lpvarOut);
}

BOOL CMultPB1Ctrl::OnGetDisplayString(DISPID dispid, CString& strValue)
{
    // Determine if we need to display a property
    // value string.
  switch (dispid)
    {

    // Display a ButtonType property value.
    case dispidButtonType:

        // Determine which value to display.
        switch (m_buttonType)
        {
        case 0:
            strValue = "0 - On/Off";
            break;
        case 1:
            strValue = "1 - Yes/No";
            break;
```

Continued

Listing 13-7: *(continued)*

```
        case 2:
            strValue = "2 - Red/Yellow/Green";
            break;
        case 3:
            strValue = "3 - User Mode";
            break;
        }

        // Return true so the programming environment
        // knows we returned a value.
        return true;
    // Display a ModalResult property value.
    case dispidModalResult:

        // Determine which value to display.
        switch (m_modalResult)
        {
        case -1:
            strValue = "-1 - User Mode";
            break;
        case 1:
            strValue = "1 - On";
            break;
        case 2:
            strValue = "2 - Off";
            break;
        case 3:
            strValue = "3 - Yes";
            break;
        case 4:
            strValue = "4 - No";
            break;
        case 5:
            strValue = "5 - Red";
            break;
        case 6:
            strValue = "6 - Yellow";
            break;
        case 7:
            strValue = "7 - Green";
            break;
        }

        // Return true so the programming environment
        // knows we returned a value.
        return true;
    }

  return COleControl::OnGetDisplayString(dispid, strValue);
}
```

The whole purpose of the OnGetPredefinedStrings( ) method is to fill two arrays with values that express the number of possible property values for a specific property. The first array, pStringArray, contains a list of plain text descriptions of the various property values. This is what the user will see when he or she clicks on the drop-down listbox in the Properties Windows. The second array, pCookieArray, contains a list of values that the client passes back to the control once the user has made a selection. You need to make the cookie values unique enough so that the OnGetPredefinedValue( ) method knows what to do with it. It's absolutely essential that you return a value of true (as shown in the code) so that the client knows that you've supplied predefined strings for the property.

The OnPredefinedValue( ) method's purpose is to translate the cookie it receives from the client into something the control can use to set the property. This requires the method to fill a poorly documented data structure. The first thing you need to do is initialize the data structure so that you can fill it with the appropriate values. As shown in the code, you must provide a minimum of two values. The first is the virtual type of the data that you're supplying. In this case, we're working with a short integer, which requires the VT_I2 constant. The second value is the property value itself. In this case, we simply can use the cookie value since I filled the cookie with the same value as the property. However, you don't absolutely follow this methodology, and it definitely won't work with things such as strings. Again, you must return a value of true when exiting the method, or the control won't know that you've performed the required translation of the cookie to a property value.

The OnGetDisplayString( ) is the simplest of the three methods. You simply need to detect what the current state of the requested property is, then return a string that reflects that status. As before, you need to return true to tell the client that you've provided a string value for the requested property. This is the optional method. Although Visual C++ doesn't require it to work with properties that include default values, products such as Visual Basic do because they have a Property Window that requests the information from your control.

The ATL Version

This section of the chapter looks at the ATL version of the component that we just created using MFC in the previous section. You'll find that many of the procedures for creating a component using ATL are completely different from those in the previous section because ATL assumes a lot less about what you need.

CD-ROM Location: Chapt12\MultPB2
Program Name: MultPB2.DLL
Source Code Modules in Text: MultPB2.CPP, MultPB2.H, MultPB2.IDL

Just to make it easier for you to compare the creation process for ATL versus MFC components, let's follow approximately the same sequence of construction steps. But the way you need to perform the steps differs. The following sections show you how to create a duplicate of the multiple state pushbutton control that we just created using ATL.

Improving the Efficiency of a Control

There are a lot of different ways to optimize the controls you create; just using ATL instead of MFC represents a major improvement in code size no matter what ATL specific methodology you use. Of course, this code size improvement comes at the cost of increased development time and potential debugging problems. Obviously, there's no one-size-fits-all solution when it comes to component technology—just tradeoffs that you need to weigh carefully.

However, there's one way to improve the efficiency of a control. Adding the IPointerInactive interface to your control allows the control to react normally when the user needs to work with it, but turn itself off when the user doesn't need it. Using this interface means that your control won't waste processor cycles when not in use. Now, here's the tradeoff you need to consider. Using the IPointerInactive interface can cause compatibility problems when a container doesn't support the interface and may prevent the control from activating itself in other environments. (In other words, it may not do anything at all to response to mouse or keyboard events.)

In addition, adding any interface to your control increases code size and development time. Increasing the efficiency of your control comes at a price, but the cost may be worth it if you need the ultimate in efficiency in your application.

Creating the ATL Version Program Shell

As with the MFC example, the first thing we need to do is create a program shell for our ActiveX control. However, in this case, we need to perform more preparatory work because ATL doesn't assume anything at all about the control you want to create. For example, the program shell doesn't include any objects, which means that you need to add everything manually. In sum, ATL requires you to perform three completely separate setup steps to get the same program shell that we started out with for the MFC example using a single procedure.

Here are the three program shell setup steps that we'll perform in this section:

1. Create the ATL program shell. This shell is quite empty; it doesn't even include any objects.
2. Add the required object to the ATL program shell. In this case, add an object based on a button, just as we did for the MFC example.
3. Add a property page object. The ATL method assumes that you don't need any property pages at all. You need to add the property pages as a separate step. Once you add the property page, setting it up is similar to the process you followed for MFC.

Now that you have a better idea of what we're going to do, let's look at the procedures for doing it. The following sections show you how to create an ATL program shell that approximates the MFC shell we created in the previous section.

The method you use to compile an ATL control is extremely important because it affects how the final control reacts. If you choose the minimum size options, then the Visual C++ compiler does everything possible to minimize control size, including using other DLLs as appropriate. On the other hand, if you use the minimum dependency option, Visual C++ does everything possible to reduce the number of external DLLs required to make the control work. However, this lack of dependency can increase the control's size. In most cases, the minimum size option is optimal for Internet download where you need to check the user's machine for other files. Otherwise, the minimum dependency option is the best choice because it ensures that the user has everything needed to use the control with a minimum number of files to download from your Web site.

Creating an ATL Program Shell

Our first task is to create an ATL program shell. The following procedure shows you how. I assume that you already have Visual C++ started and that you use Version 6.0 of the product.

Try This

1. Use the File | New command to display the New dialog box.
2. Highlight the ATL COM AppWizard entry, type **MultPB2** in the Project name field, then click OK. You should see the ATL COM AppWizard - step 1 of 1 dialog box shown in Figure 13-20. We need to use the DLL format for this control — even though an ActiveX control normally uses an OCX extension, it's really a DLL. In addition, we want to combine the proxy and stub code so that the resulting file consumes just one DLL instead of two.

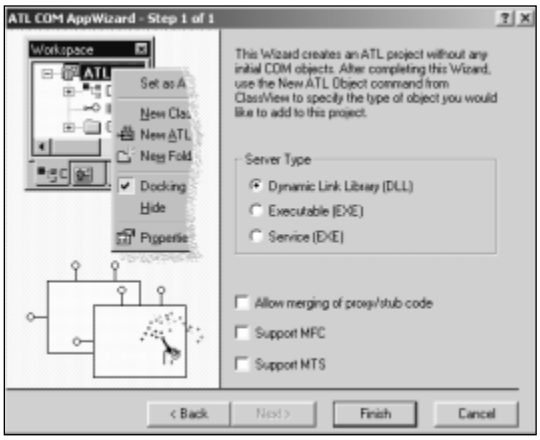

ATL COM AppWizard - Step 1 of 1
Workspace
This Wizard creates an ATL project without any initial COM objects. After completing this Wizard, use the New ATL Object command from ClassView to specify the type of object you would like to add to this project.
Server Type
Dynamic Link Library (DLL)
Executable (EXE)
Service (EXE)
Allow merging of proxy/stub code
Support MFC
Support MTS
< Back
Next >
Finish
Cancel

Figure 13-20: The ATL COM AppWizard - step 1 of 1 dialog box allows you to perform some preliminary configuration of your control.

3. Check the Allow merging of proxy/stub code option. Choosing this option will allow Visual C++ to create a single control.
4. Click Finish. You should see the New Project Information dialog box shown in Figure 13-21.
5. Click OK. Visual C++ creates the empty ATL project shell for you.

Figure 13-21: The New Project Information dialog box shows the settings for the ATL control.

Adding an Object to the Program Shell

All of the extra work you have to do to create an ATL shell pays off in one respect, design flexibility. When you create an ActiveX component shell using MFC, there's a single field on one of the wizard dialog boxes that allows you to choose how many objects the component contains. If you change your mind later, you have to go through a lot of extra work to add another object and forget about removing an object without a lot of hair pulling. ATL makes life easier by allowing you to add objects as needed, when you decide that you need them. Removing an object may be a little harder, but it's not nearly as difficult as when you use MFC. The following procedure shows you how to add an ATL ActiveX object to the program shell we created in the previous section.

Try This

1. Right-click the MultPB2 Classes folder on the ClassView tab of the Workspace Window; then choose New ATL Object from the context menu. You should see the ATL Object Wizard dialog box.
2. Highlight the Controls entry in the Category list. You'll see a list of ATL-based ActiveX objects like the ones shown in Figure 13-22. We discussed the interfaces that each of these object types requires in the "Understanding the

ActiveX Interfaces" section of this chapter. (See the sidebar entitled "Choosing an ATL ActiveX Object Type" for further details on which object type to use for specific situations.)

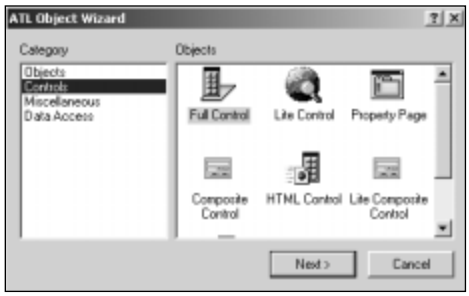


Figure 13-22: ATL enables you to choose from a wide variety of object types to begin creating your ActiveX control.

3. Highlight Full Control, then click Next. You should see the ActiveX Object Wizard Properties dialog box shown in Figure 13-23. The Names tab is where you determine things like the object's class name and the files used to support it.

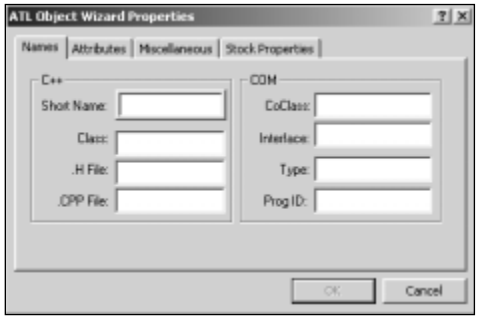


Figure 13-23: The Names tab allows you to choose the class name for your new control.

4. Type **MultPB2Ctrl** in the Short Name field. Visual C++ automatically adds default entries to the remaining fields for you. However, the name in the Type field isn't very descriptive, and that's what gets displayed in the various control selection dialog boxes.
5. Type **Multiple State Pushbutton Control (ATL)** in the Type field. Changing this entry makes it easier for someone to find the control in a list and reduce the chance that someone chooses it by mistake.
6. Click the Attributes tab and you'll see a list of control attributes like the ones shown in Figure 13-24. This is where you choose the characteristics of the object. In most cases, these settings are just fine. However, since we want to support events, we need to add connection point support to the control.

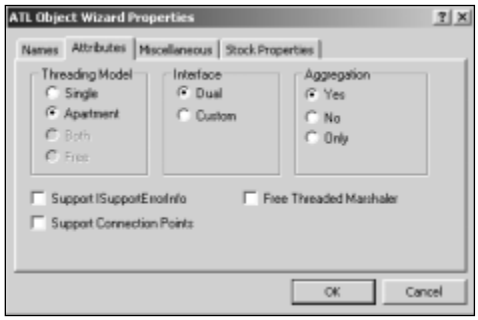


Figure 13-24: The Attributes tab allows you to choose functionality options such as what threading model the control uses and whether it supports connection points.

7. Check the Support Connection Points option. This will allow us to add events to the control later.

You can reduce the size of the resulting control by setting Aggregation to No. However, this means that the control can't be used as a basis for creating other controls. If you choose the Only option, then this control can be used only as a basis for creating other controls—not by itself. Adding the ISupportErrorInfo interface will allow your control to provide error information to the client and to detect client error conditions in some cases. However, the client also has to support this interface before you can do anything about errors—something that most clients don't support.

8. Click the Miscellaneous tab. This tab contains a variety of configuration options for the control such as what general control it's derived from, as shown in Figure 13-25. Notice that this tab also includes drawing options such as whether the control always needs to appear within a window. It's important to know what kinds of features the environment that the control appears in supports.

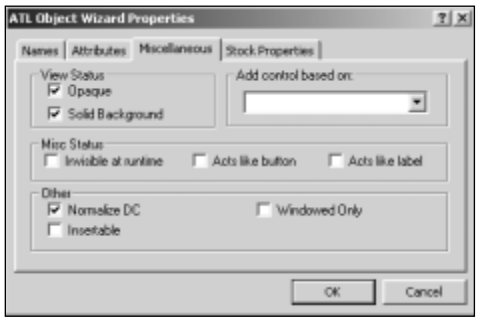


Figure 13-25: The Miscellaneous tab allows you to configure control features such as the general control it subclasses and any drawing options.

9. Check the Acts like button option and choose Button in the Add control based on listbox. The Acts like button option allows the control to become a default control in a dialog box (the one that gets selected automatically if the user presses Enter—it also gets displayed with a thicker line around the outside of the button). The Add control based on listbox determines what general control is used as a basis for creating this control. Notice that Visual C++ automatically selects the Windowed Only option for you, which forces the pushbutton always to appear in a Window (allowing it full access to mouse and other user interface events).
10. Click the Stock Properties tab. You'll see a list of stock properties that ATL supports as shown in Figure 13-26. This is where you determine what stock properties get added to your control, although you can always add them later. In this case, we add the same stock properties that we added for the MFC version of the control.

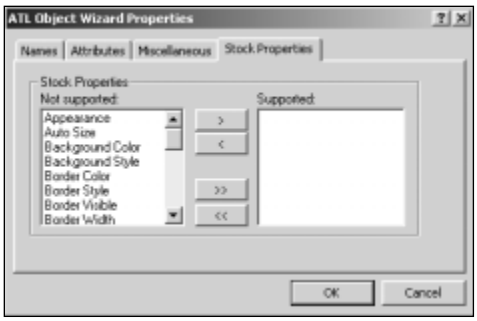


Figure 13-26: The Stock Properties tab allows you to choose which stock properties get added to your control during creation.

ATL doesn't provide any method for selecting either stock methods or events during control creation. We need to add these features later in the section using the same procedures we use to add custom properties, methods, and events. Hopefully, Microsoft will correct this oversight in a future version of ATL.

11. Highlight the Caption, Enabled, and Font properties, then click the > (right arrow) button. The stock properties get added to the Supported list on the right side of the dialog box.
12. Click OK. Visual C++ creates the new control for you.

Choosing an ATL ActiveX Object Type

Visual C++ provides a variety of ATL-based ActiveX object types, each of which is designed to perform a specific task. Providing multiple object types gives you greater flexibility in creating ActiveX controls because you can choose which interfaces to support at the outset rather than add them later. In short, ATL actually provides one optimization that the MFC ActiveX Control Wizard doesn't. The following list provides you with some details about each of the ATL-based ActiveX objects.

- **Full Control:** The most complete generic control you can create with ATL. It's designed to support all of the interfaces required by every type of container. Use the Full Control object to create any general ActiveX control that you intend to use on both the desktop and within Internet Explorer. Of course, you also pay a size penalty for including all of the required interfaces.
- **Composite Control:** A special-purpose control that works as a container for other controls. In other words, you use this control to create an environment used to host other controls. In all other ways, this control is like the Full Control.
- **HTML Control:** Another special-purpose control used to create a DHTML-based user interface. In other words, you can theoretically use this type of control to create a miniature Internet Explorer environment. The most common use for this control is as an add-on to applications that work with Web pages most of the time. You can also use it as part of a strategy for implementing an HTML-based help desk or for displaying Web resources such as graphics.
- **Lite Control, Lite Composite Control, and Lite HTML Control:** All three of these object types work just like their standard counterparts. The main difference is the number of interfaces they support. You can use these controls only within Internet Explorer. What that means to you as a programmer is that you can use these object types to create small versions of your controls for very limited distribution.

Adding a Property Page to the Program Shell

At this point, we have an object to work with, but no property page to change the object properties with when working in environments that don't support a Property Window. This section shows you how to add a custom property page to the project. You need to use this manual procedure for any property page you want to add.

Try This

1. Right-click the MultPB2 Classes folder on the ClassView tab of the Workspace Window; then choose New ATL Object from the context menu. You should see the ATL Object Wizard dialog box.

2. Highlight the Controls entry in the Category list. You can see a list of ATL-based ActiveX objects like the ones shown in Figure 13-22.

3. Highlight Property Page, then click Next. You'll see the ActiveX Object Wizard Properties dialog box shown in Figure 13-27. The Names tab is where you determine things such as the object's class name and the files used to support it. Notice that this version of the ActiveX Object Wizard dialog box contains fewer tabs and customization options than the one shown in Figure 13-23.

Figure 13-27: A Property Page object offers different configuration features than the Full Control object.

4. Type **MultPB2PropPage** in the Short Name field. Visual C++ automatically enters values for the rest of the fields for you. In this case, the default entries work fine (though you can change the Type field if you want something more descriptive for the property page).

5. Click the Strings tab to see a dialog box like the one shown in Figure 13-28. This dialog box contains the strings that give the property page a title and tells the client where to find the property page's help file. In this case, we give the property page the same title as its MFC counterpart and won't assign a help file to it. The Doc String field provides balloon help for the property page, but most programming languages don't use it.

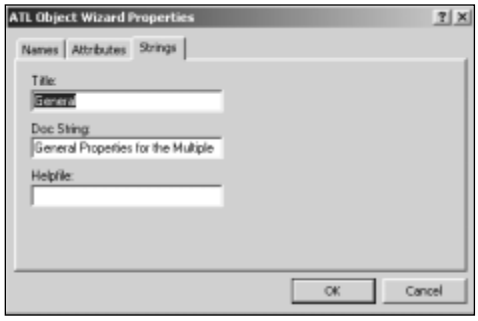


Figure 13-28: The Strings tab allows you to give your property page a name.

6. Type **General** in the Title field and General Properties for the Multiple State Pushbutton Control in the Doc String field.
7. Clear the Helpfile field.
8. Click OK. Visual C++ creates the required files and resources for the property page object. This includes creating a dialog box that you can use to add controls to the property page.

Adding Methods, Properties, and Events to the ATL Version

There's no MFC ClassWizard dialog box to use when you work with ATL. That's because you're not relying on the MFC libraries to create the control. Obviously, this also means that you need to use a different technique when adding methods, properties, and events to your control.

All of the stock properties got added to the control as part of the creation process, so we don't need to do that again. However, we still need to add the ModalResult and ButtonType custom properties. The following procedure shows you how.

Try This

1. Right-click the IMultPB2Ctrl interface on the ClassView tab of the Workspace window; then choose Add Property from the context menu. You should see the Add Property to Interface dialog box shown in Figure 13-29. This dialog box allows you to define a property name, its return value, and any arguments required for using it.

Figure 13-29: The Add Property to Interface dialog box allows you to add new properties to your control.

2. Choose short in the Property Type field. Notice that Visual C++ automatically adds the appropriate value to the Implementation field. This field allows you to see what the resulting property implementation looks like in the IDL (interface description language) file.

3. Type **ModalResult** in the Property Name field. At this point, we have the property defined, but notice that the property description shown in the Implementation field is less than helpful.
4. Click Attributes. You can see the Edit Attributes dialog box shown in Figure 13-30 (this dialog box is already filled out). This is where you can change property attributes like the help string associated with the property. There are a couple of changes we can make here. For one thing, we can create a define for the display identifier of the property so that we can use it instead of a number within the code. In addition, we need to set some other attributes for this property so that the client fully understands how to interact with it.
5. Type **"Control Return Value"** in the helpstring field. Make sure you include the double quotes around the value.
6. Choose bindable in the first free Attribute Name field, and requestedit in the second free Attribute Name field. Your Edit Attributes dialog box should look like the one shown in Figure 13-30. The bindable attribute tells the client to notify the component any time this property value changes. The requestedit attribute tells the client that the control supports the OnRequestEdit notification. In essence, the client asks the control for permission before it makes a change to a property.

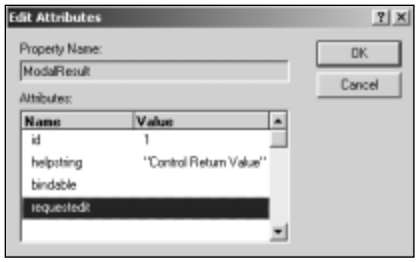


Figure 13-30: The Edit Attributes dialog box allows you to change the attributes associated with a method or property.

7. Click OK twice. Visual C++ adds the requested property to your IDL file.
8. Repeat Steps 1 through 7 to add the ButtonType property. Make sure you add an appropriate helpstring value like "Selects the Type of Button Created." Also change the property name as appropriate.

Simply adding the properties as shown in the previous procedure doesn't make those properties either persistent or even usable. Unlike MFC, where a lot of the work is done for you, ATL requires several manual entries to make a property fully usable. We see what these additional entries are in the "Defining Get/Put Methods for the Properties" section of this chapter.

Adding the methods is similar to adding properties. However, in this case, you need to add stock as well as custom methods to the object since there isn't any way to add them during object creation. The following procedure shows you how to add the methods for this example to the control.

Try This

1. Right-click the IMultPB2Ctrl interface folder on the ClassView tab of the Workspace window; then choose Add Method from the context menu. You should see the Add Method to Interface dialog box shown in Figure 13-31.

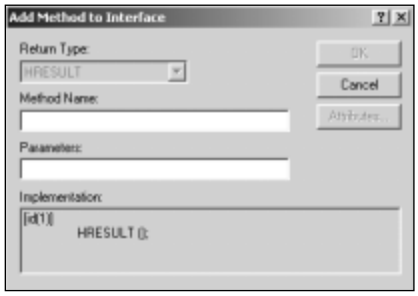


Figure 13-31: The Add Method to Interface dialog box allows you to add new methods to your control.

Notice that the Add Method to Interface dialog box doesn't allow you to change the Return Type field. This means that your method provides an HRESULT value as a return value, which isn't what the MFC version of the control does. There are two ways to handle this difference. Either you can hand edit the IDL file later to change the return type of your method or you can change the method so that it returns a value using one of the variables in the parameter list. Using the [out, retval] directional parameter allows the function to act as if you had assigned a particular return value to the method.

2. Type **SetUser** in the MethodName field.
3. Type **[in] BSTR bstrUserName** in the Properties field. You must provide both a variable type and a variable name for each parameter that you want to pass to your method. In addition, you can provide optional information such as the [in] directional parameter for this method that tells Visual C++ that the bstrUserName argument is passed from the client to the control.
4. Click OK. Visual C++ adds the method to your application.
5. Perform Steps 1 through 4 for the DoClick( ), DoChange( ), and GetUser( ) methods. You don't need to provide any value for the Properties field for either the DoClick( ) or DoChange( ) methods. The GetUser( ) method requires a Properties field entry of [out, retval] BSTR* bstrUserName.

Now that you've added the methods and properties required for this example, it's time to add the OnClick( ) event. Events are handled through a different interface than the one we've used so far. You need to work with the IMultPB2CtrlEvents interface in this case, though the actual process of adding the event is the same as adding a method to your control. The following procedure shows you how.

Try This

1. Right-click the _IMultPB2CtrlEvents interface folder on the ClassView tab of the Workspace window; then choose Add Method from the context menu. You should see the Add Method to Interface dialog box shown in Figure 13-31. Notice that you can choose a return value for an event method, unlike a standard control method. This is the one difference between the two — but a very important one.
2. Choose void in the Return Type field. The Click( ) event doesn't need to return a value of any kind.
3. Type **Click** in the MethodName field. The actual event name is Click, not OnClick, which is the name of the function used to handle the Click event. This is another place where you need to keep the differences between events and standard control methods in mind.

We didn't add any properties to the Click event because it doesn't require any. However, you need to keep properties in mind when you implement other kinds of events. For example, the KeyDown event requires two input parameters: one for the key code and a second for the shift status. Make sure you understand what parameters are required for stock events before you try to implement them. Obviously, custom event parameters rely on what you want the event to do.

4. Click OK. Visual C++ adds the event method to your application.

Now, the MFC version of our control has properties, methods, and events implemented in shell form. Just look at the IMultPB2Ctrl interface folder below the CMultPB2Ctrl folder in ClassView and you'll see a list of property and method implementation methods. On the other hand, though our ATL control does have the property and method shell entries made, it doesn't have anything more than the IDL entries made for the Click event.

Fortunately, you don't have to add all of the event method entries required to implement the _IMultPB2CtrlEvents interface manually. What you need to do is implement the event interface that we've created within the CMultPB2Ctrl object. Simply right-click the CMultPB2Ctrl folder within the ClassView tab of the Workspace window and choose Implement Connection Point from the context menu. You should see an Implement Connection Point dialog box like the shown in Figure 13-32. Check the interface that you want to implement (there should be only one in this case), then click OK. Added is a new proxy class named CProxy_IMultPB2CtrlEvents<class T> with a single method named Fire_Click( ), which allows you to output Click events to the client. You also can use this technique to implement interfaces other than the custom interface created specifically for the control.

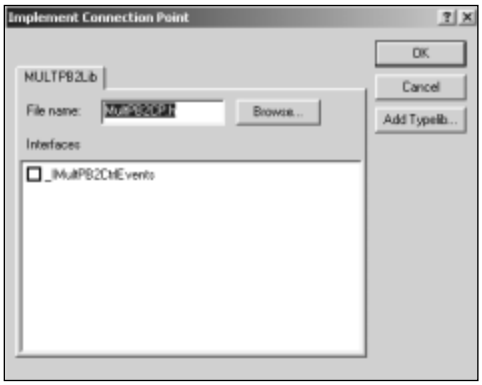


Figure 13-32: Use the Implement Connection Point dialog box to implement the events for your control in a proxy class.

Designing Resources for the ATL Version

Many of the resources you need for this version of the Multiple State Pushbutton Control are the same as the ones we created for the MFC version. This includes control bitmap, the About dialog box and associated icon, the version information, and the control layout of the custom property page. Since I already discussed these resources, I won't mention them again here. However, there are some coding differences that I will mention when we begin adding code to the control.

Make absolutely certain you add all of the required resources or some of the code in the following sections may not work as anticipated. Use the MFC version of the control as a template for adding the resources required for the ATL version. Change all MultPB1 references to MultPB2 for this example.

Writing the ATL Version Code

This section shows you how to add the code required to make the ATL version of our Multiple State Pushbutton Control functional. You may notice quite a few similarities between this code and the code used for the MFC version. However, it's the differences that are the most important. ATL requires making changes in the way you handle some of the method coding. There are also additional requirements for making things such as the About dialog box functional that you don't need to worry about with the MFC version. The following sections describe each of the major code segments for this version of the control.

Coding the About Dialog Box

When we created the MFC version of the Multiple State Pushbutton, the only thing we needed to do was add the dialog resource elements required to make the About dialog box functional. Unlike MFC, ATL doesn't assume that you want an About dialog box for your control. What this means to you as a programmer is that you need to add the class and required code to make an About dialog box functional. That's what this section shows you.

In some cases, you really don't need an About dialog box for an ActiveX control. For example, if the control you create is only for in-house corporate or personal use, you don't need to add an About dialog box. In fact, since the About dialog box does consume space, you may be better off leaving it out unless you really do need it. There are times (like when you're dealing with an intranet site) that even a small reduction in code size can make a difference in the time required to download the control.

Before we can add some code to make the About dialog box function, we need to perform some ATL specific setup. The following steps show you how to create the class and method required to access the About dialog box.

Try This

1. Right-click the MultPB2 Classes folder in the ClassView tab of the Workspace Window; then choose New ATL Object from the context menu.
2. Choose the Miscellaneous option in the Category field. You should see a list of miscellaneous ATL objects that you can create, as shown in Figure 13-33.

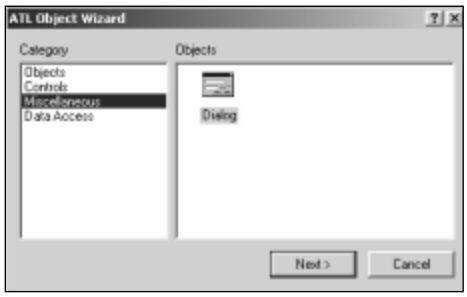


Figure 13-33: The Miscellaneous ATL object category includes a Dialog object.

3. Highlight the Dialog object, then click Next. You should see an ATL Object Properties dialog box containing a single tab as shown in Figure 13-34. The only thing we really need to do, in this case, is define the short name for the About dialog box class.

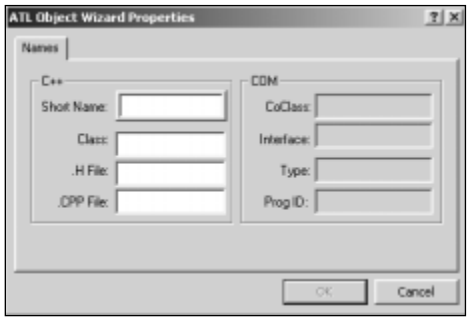


Figure 13-34: Dialog boxes require only one page of property settings.

4. Type **AboutDialog** in the Short Name field. Visual C++ automatically adds default entries to the other fields for you.
5. Click OK. Visual C++ adds the new class for you, which includes a dialog box resource that looks like a standard dialog box, rather than an About dialog box. I reveal how to change this later in the section. Now it's time to add a method to the CMultPB2Ctrl class.
6. Right-click the IMultPB2Ctrl interface folder, then choose Add Method from the context menu. You should see an Add Method to Interface dialog box like the one shown in Figure 13-31.
7. Type **AboutBox** in the Method Name field, then click OK. Visual C++ adds the requested method to your control.

Now that we have an CAboutDialog class and an AboutBox( ) method to access it with, there are three programming tasks required to make the About dialog box functional. Listing 13-8 shows the code you need to add to the AboutBox( ) method, which is the first step you need to take.

Listing 13-8: **AboutBox( ) Method Source Code**

```
STDMETHODIMP CMultPB2Ctrl::AboutBox()
{
    // Create an instance of the CAboutDialog class.
    CAboutDialog dlgAbout;

    // Display the dialog.
    dlgAbout.DoModal();

    // Return a non-error result.
    return S_OK;
}
```

As you can see, the AboutBox( ) method code is quite simple. We simply need to create an instance of the CAboutDialog class and use the DoModal( ) method to display it onscreen. It resembles the code that the MFC version of the control uses. The only difference is that you need to add this code to the ATL control manually.

Adding the CAboutDialog class to your project doesn't automatically make it available to the CMultPB2Ctrl.CPP source file. You have to include the CAboutDialog class header in the CMultPB2Ctrl.H file just as you would for any external source file. Here's the code you need to add in this case.

```
// Add support for the About Box.
#include "AboutDialog.h"
```

Finally, you want programming languages that include a Properties Window (such as Visual Basic) to display an About entry. Clicking this entry displays the About dialog box that we've created. In most cases, the IDEs for these other languages look for a specific dispatch identifier (DISPID), so you need to change the DISPID for the About dialog box in the MultPB2.IDL file as shown below in bold.

```
[id(DISPID_ABOUTBOX), helpstring("Displays the About dialog")]
HRESULT AboutBox();
```

Make sure you use the correct DISPID value and change the helpstring value. The DISPID value gets the About entry in the Properties window, whereas the helpstring value displays helpful text at the bottom of the Properties window when the user selects a specific property.

Creating Persistent Properties

Making properties persistent when working with ATL totally differs from what you did with MFC. In fact, you make the required entries in the MultPB2Ctrl.H file instead of the CPP file. Listing 13-9 shows the code you need to add in bold.

Listing 13-9: **Making ATL Properties Persistent**

```
BEGIN_PROP_MAP(CMultPB2Ctrl)
    PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
    PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
    PROP_ENTRY("Caption", DISPID_CAPTION, CLSID_NULL)
    PROP_ENTRY("Enabled", DISPID_ENABLED, CLSID_NULL)
    PROP_ENTRY("Font", DISPID_FONT, CLSID_StockFontPage)
    // Add the custom properties.
    PROP_ENTRY("ModalResult", DISPID_MODALRESULT, CLSID_NULL)
    PROP_ENTRY("ButtonType", DISPID_BUTTONTYPE, CLSID_NULL)
```

These entries are relatively easy to understand. The PROP_ENTRY( ) macro takes three arguments. The first is the text name of the property as it appears in a Properties Window. The second is the dispatch identifier of the property. The third entry specifies a special dialog box, if any, used to configure the property. Notice that the Font entry contains an entry of this sort — the stock font property page. If the property doesn't require any special configuration, then you need to supply the CLSID_NULL value as shown.

You may wonder where I got the dispatch identifiers shown in Listing 13-9. I defined them in the MultPB2.IDL file as shown in Listing 13-10 in bold. There isn't any requirement to create these entries, but doing so makes your code a lot easier to understand. If you don't create the DISPID values, then you need to use the

numbers that ATL originally supplied (which means you need to keep some kind of reference for those numbers handy). Notice that I also replaced the id values for the custom properties with their custom DISPID value counterparts for consistency.

Listing 13-10: **Using Custom DISPID Values**

```
// Create some defines for the various dispatch IDs.
const int DISPID_MODALRESULT    = 1L;
const int DISPID_BUTTONTYPE     = 2L;
const int DISPID_SETUSER        = 3L;
const int DISPID_GETUSER        = 4L;
const int DISPID_DOCHANGE       = 6L;

// Properties
[propputref, id(DISPID_FONT)]
    HRESULT Font([in]IFontDisp* pFont);
[propput, bindable, requestedit, id(DISPID_FONT)]
    HRESULT Font([in]IFontDisp* pFont);
[propget, id(DISPID_FONT)]
    HRESULT Font([out, retval]IFontDisp** ppFont);
[propput, id(DISPID_ENABLED)]
    HRESULT Enabled([in]VARIANT_BOOL vbool);
[propget, id(DISPID_ENABLED)]
    HRESULT Enabled([out,retval]VARIANT_BOOL* pbool);
[propput, id(DISPID_CAPTION)]
    HRESULT Caption([in]BSTR strCaption);
[propget, id(DISPID_CAPTION)]
    HRESULT Caption([out,retval]BSTR* pstrCaption);
[propget, id(DISPID_MODALRESULT),
    helpstring("Control Return Value"),
    bindable, requestedit]
    HRESULT ModalResult([out, retval] short *pVal);
[propput, id(DISPID_MODALRESULT),
    helpstring("Control Return Value"),
    bindable, requestedit]
    HRESULT ModalResult([in] short newVal);
[propget, id(DISPID_BUTTONTYPE),
    helpstring("Selects the Type of Button Created"),
    bindable, requestedit]
    HRESULT ButtonType([out, retval] short *pVal);
[propput, id(DISPID_BUTTONTYPE),
    helpstring("Selects the Type of Button Created"),
    bindable, requestedit]
    HRESULT ButtonType([in] short newVal);
```

Some of the IDL entries you make for a property won't show up in Visual C++, but are important if you plan to use the ActiveX control in another programming environment such as Visual Basic. The helpstring entry appears at the bottom of the Properties window in IDEs that support this feature. It helps the user understand what the property is used for. Obviously, the usefulness of this feature is directly proportional to the amount of text you provide. Don't write a book about the property—the helpstring should be short and to the point. What the helpstring should do is provide the user with enough information to make a decision about the property setting. In most cases, you may want to define helpstrings for both custom and stock properties. The ATL wizard provides only a very generic helpstring for custom properties and nothing at all for stock properties.

Defining Get/Put Methods for the Properties

Unlike the MFC version of our control, the ATL version uses Put, instead of Set methods, to change the current value of a property. Fortunately, this is a difference in naming only, and not a real difference in how the method is implemented. However, there are enough differences that you need to keep in mind when porting your code from one environment to the other.

Begin by defining the same series of member variables that we defined for the MFC version of the control as shown Listing 13-11. Notice that there are some differences from the MFC version because ATL doesn't support MFC classes such as CString and CArray. Since aDisplayValues is now a static array, we need to include one additional variable for tracking the aDisplayValues array elements in use named iDisplaySize. Since I already discussed these variables, I won't discuss them again in this section. Note that I did retain the variable names from the MFC version of the control to make it easier for you to track the changes in variable usage.

Listing 13-11: **Get/Put Method Variable Listing**

```
int iCurrentChoice;
int iDisplaySize;
LPTSTR oUserString;
LPTSTR aDisplayValues[10];
short m_modalResult;
short m_buttonType;
```

Once you have some variables to use, you can add the code required to make the Get/Put methods work as shown in Listing 13-12. Though the methodology of this code should look familiar, you should also see some definite differences between this code and the MFC version in Listing 13-2.

Listing 13-12: **ModalResult and ButtonType Get/Put Method Source Code**

```
STDMETHODIMP CMultPB2Ctrl::get_ModalResult(short *pVal)
{
    // Return the stored value.
    *pVal = m_modalResult;

    // Return a non-error message
    return S_OK;
}

STDMETHODIMP CMultPB2Ctrl::put_ModalResult(short newVal)
{
    // Tell Visual C++ that we'll be converting BSTR values.
    USES_CONVERSION;
    // Set the ModalResult value.
    m_modalResult = newVal;

    // Determine which ModalResult has been
    // selected, then set the dependent properties
    // and member variables to match. Make sure you
    // set all of the dependent properties--even
    // stock properties.
    switch (newVal)
    {
    case 1:
        m_ctlButton.SetWindowText("On");
        m_bstrCaption = T2BSTR("On");
        iCurrentChoice = 0;
        m_buttonType = 0;
        break;
    case 2:
        m_ctlButton.SetWindowText("Off");
        m_bstrCaption = T2BSTR("Off");
        iCurrentChoice = 1;
        m_buttonType = 0;
        break;
    case 3:
        m_ctlButton.SetWindowText("Yes");
        m_bstrCaption = T2BSTR("Yes");
        iCurrentChoice = 0;
        m_buttonType = 1;
        break;
    case 4:
        m_ctlButton.SetWindowText("No");
        m_bstrCaption = T2BSTR("No");
        iCurrentChoice = 1;
        m_buttonType = 1;
```

```
        break;
    case 5:
        m_ctlButton.SetWindowText("Red");
        m_bstrCaption = T2BSTR("Red");
        iCurrentChoice = 0;
        m_buttonType = 2;
        break;
    case 6:
        m_ctlButton.SetWindowText("Yellow");
        m_bstrCaption = T2BSTR("Yellow");
        iCurrentChoice = 1;
        m_buttonType = 2;
        break;
    case 7:
        m_ctlButton.SetWindowText("Green");
        m_bstrCaption = T2BSTR("Green");
        iCurrentChoice = 2;
        m_buttonType = 2;
        break;
    case -1:
        m_ctlButton.SetWindowText("Button");
        m_bstrCaption = T2BSTR("Button");
        iCurrentChoice = 0;
        m_buttonType = 3;
        break;
    }

// Determine which button the user has selected, then
// fill the display array with an appropriate set of
// display values.
switch (m_buttonType)
{
case 0:
   strcpy(aDisplayValues[0], "On");
   strcpy(aDisplayValues[1], "Off");
   iDisplaySize = 2;
   break;
case 1:
   strcpy(aDisplayValues[0], "Yes");
   strcpy(aDisplayValues[1], "No");
   iDisplaySize = 2;
   break;
case 2:
   strcpy(aDisplayValues[0], "Red");
   strcpy(aDisplayValues[1], "Yellow");
   strcpy(aDisplayValues[2], "Green");
     iDisplaySize = 3;
   break;
case 3:
```

Continued

Listing 13-12: *(continued)*

```
      iDisplaySize = 1;
      if (strlen(oUserString) > 0)
         SetUser(T2BSTR(oUserString));
   }

    // Register the update
   SetDirty(true);
   FireViewChange();

   // Return a non-error message
   return S_OK;
}

STDMETHODIMP CMultPB2Ctrl::get_ButtonType(short *pVal)
{
    // Return the stored value.
    *pVal = m_buttonType;

    // Return a non-error message
    return S_OK;
}

STDMETHODIMP CMultPB2Ctrl::put_ButtonType(short newVal)
{
    // Tell Visual C++ that we'll be converting BSTR values.
    USES_CONVERSION;
    // Set the ButtonType value.
    m_buttonType = newVal;

    // Determine which ButtonType has been
    // selected, then set the dependent properties
    // and member variables to match.
    switch (newVal)
    {
    case 0:
        m_ctlButton.SetWindowText("On");
        m_bstrCaption = T2BSTR("On");
        m_modalResult = 1;
        break;
    case 1:
        m_ctlButton.SetWindowText("Yes");
        m_bstrCaption = T2BSTR("Yes");
        m_modalResult = 3;
        break;
    case 2:
        m_ctlButton.SetWindowText("Red");
        m_bstrCaption = T2BSTR("Red");
        m_modalResult = 5;
```

```
        break;
    case 3:
        m_ctlButton.SetWindowText("Button");
        m_bstrCaption = T2BSTR("Button");
        m_modalResult = -1;
        break;
    }

    // Set the currently selected multi-state
    // option value.
    iCurrentChoice = 0;

   // Determine which button the user has selected, then
   // fill the display array with an appropriate set of
   // display values.
   switch (m_buttonType)
   {
   case 0:
      strcpy(aDisplayValues[0], "On");
      strcpy(aDisplayValues[1], "Off");
      iDisplaySize = 2;
      break;
   case 1:
      strcpy(aDisplayValues[0], "Yes");
      strcpy(aDisplayValues[1], "No");
      iDisplaySize = 2;
      break;
   case 2:
      strcpy(aDisplayValues[0], "Red");
      strcpy(aDisplayValues[1], "Yellow");
      strcpy(aDisplayValues[2], "Green");
        iDisplaySize = 3;
      break;
   case 3:
      iDisplaySize = 1;
      if (strlen(oUserString) > 0)
         SetUser(T2BSTR(oUserString));
   }

   // Register the update
   SetDirty(true);
   FireViewChange();

   // Return a non-error message
   return S_OK;
}
```

As you can see, the basic idea of setting and getting control properties is the same when using ATL; it's the actual implementation that varies. In this case, we're working with LPTSTR values and standard arrays rather than using the CString and CArray classes as we did in the MFC example. These changes account for a large part of the differences between the code shown here and the code in the MFC section.

Notice that there are a couple of additions to the MFC code. For one thing, you need to use the SetDirty( ) method to tell the control to update its property values. In addition, you should use the FireViewChange( ) method to ensure the control gets repainted to show any modifications to its appearance. Finally, since ATL doesn't perform any automatic conversion of BSTR variables for you, the code needs to perform this task manually (as shown in the example code). Make sure that you include the USES_CONVERSION macro at the top of any methods performing a conversion from BSTR to C-specific data type. Visual C++ normally provides an ambiguous error message if you don't include the macro, so checking for the USES_CONVERSION macro is one way to ensure that your code compiles.

Adding Code for the Control's Methods

You may find that implementing a method under ATL — and its required steps — closely resembles the same method under MFC. The main difference that you'll see is in how those steps are implemented. For example, instead of returning a result directly as we did in MFC, you return an HRESULT value and place the actual return value for the method in one of the variables passed by the client.

There will be other differences as well. ATL doesn't maintain the state of your control automatically as MFC does. This means the method includes additional state change code to ensure the control works as anticipated.

Finally, the lack of MFC class support means that you need to use standard C data types in many cases. The example code in this section shows just one way to get around the lack of class-support problem. Obviously, the methods you use will be determined as much by the level of portability your control requires as the resources you can expect on the client machine. This is one area where you can quickly inflate the size of a control to the point of being unusable on the Internet.

The specification for our control includes four methods: DoChange( ), DoClick( ), GetUser( ), and SetUser( ). Listing 13-13 shows the source code for all four methods. Make sure you take the time to compare the code in this section with the code we created earlier for the MFC example.

Listing 13-13: DoChange(), DoClick(), SetUser(), and GetUser() Methods Source Code

```
STDMETHODIMP CMultPB2Ctrl::SetUser(BSTR bstrUserString)
{
  LPTSTR  oSingleEntry;   // A single entry.
  LPTSTR  lpszTemp;       // A temporary whole string holder.

  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Set the user string to a new value.
  strcpy(oUserString, OLE2T(bstrUserString));
  lpszTemp = _strdup(oUserString);

  // Set the current choice.
  iCurrentChoice = 0;

  // Make sure we have something to display.
  if((strlen(lpszTemp)==0) || (strchr(lpszTemp, ';')==NULL))
  {
    // If not, then set all of the require user mode
    // values back to their default state.
    m_ctlButton.SetWindowText("Button");
    m_modalResult = -1;

    // Return without parsing the string.
    return S_OK;
  }

  // Set the display size.
  iDisplaySize = 1;

  // Find the first token.
  oSingleEntry = (LPTSTR) CoTaskMemAlloc(80);
  strncpy(oSingleEntry, lpszTemp, strcspn(lpszTemp, ";"));
  lpszTemp = strpbrk(lpszTemp, ";");

  // Parse the string and create the display array.
  while (!strchr(lpszTemp, ';') == NULL)
  {
    // Add a new entry to the display array.
    strcpy(aDisplayValues[iDisplaySize - 1], oSingleEntry);
    // Reposition the string pointer.
    if (strlen(lpszTemp) > 0)
      lpszTemp++;
    if((strlen(lpszTemp)>0) && (!strchr(lpszTemp, ';')==NULL))
    {
```

Continued

Listing 13-13: *(continued)*

```
      // Load the next token into the single entry variable.
      strncpy(oSingleEntry, lpszTemp, strcspn(lpszTemp, ";"));
      // Increment the display value array size if necessary.
      iDisplaySize++;
      // Set the user string to the next token.
      lpszTemp = strpbrk(lpszTemp, ";");
    }
  }

  // Set the caption to the first user mode string value.
  m_ctlButton.SetWindowText(aDisplayValues[0]);
  // Set ModalResult to 1 so the client can process the // results.
m_modalResult = 1;
  // Register the update
  SetDirty(true);
  FireViewChange();

  return S_OK;
}

STDMETHODIMP CMultPB2Ctrl::GetUser(BSTR* bstrUserString)
{
  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Return the current user string value.
  *bstrUserString = CComBSTR(oUserString);

  return S_OK;
}

STDMETHODIMP CMultPB2Ctrl::DoClick()
{
  // Determine what the next button position is.
  if (iCurrentChoice < iDisplaySize - 1)
  {
    iCurrentChoice++;
    m_modalResult++;
  }
  else
  {
    m_modalResult = m_modalResult - iCurrentChoice;
    iCurrentChoice = 0;
  }

  // Display the button text for the current position.
  if (iDisplaySize > 0)
    m_ctlButton.SetWindowText(aDisplayValues[iCurrentChoice]);
  // Perform the default action.
```

```
  Fire_Click();

  return S_OK;
}

STDMETHODIMP CMultPB2Ctrl::DoChange()
{
  // Determine what the next button position is.
  if (iCurrentChoice < iDisplaySize - 1)
  {
    iCurrentChoice++;
    m_modalResult++;
  }
  else
  {
    m_modalResult = m_modalResult - iCurrentChoice;
    iCurrentChoice = 0;
  }

  // Display the button text for the current position.
  if (iDisplaySize > 0)
    m_ctlButton.SetWindowText(aDisplayValues[iCurrentChoice]);
  return S_OK;
}
```

The first thing you may wonder is why I created two copies of the input string for the SetUser ( ) method. The main reason is that using two variables allows the code to process the string without modifying the saved value. Notice that the processing code actually changes the value of the temporary string using the strpbrk( ) function. Using this function allows the code to create the tokens required to process each button value passed by the user. Since this method does work with a control property, you may want to keep the variable used to hold the property value separate from the rest of the processing mechanism.

The overall processing strategy for this version of the SetUser( ) method is the same as the MFC version. The method checks for input from the user. If the input is either incomplete or missing, then the method resets the button properties and exits. On the other hand, if there's data to process, the method uses the oSingleEntry variable to hold the current token and the lpszTemp variable to hold the unprocessed tokens. Processed tokens are lost once you place them in the aDisplayValues array.

The DoClick( ) and DoChange( ) methods are essentially the same as their MFC counterparts, so I won't discuss them in this section. The main difference between the two methods is that DoClick( ) fires a click event once it changes the button display.

Defining the OnClick( ) Event

The OnClick( ) event requires some special monitoring when working with ATL. You need to monitor the user's mouse clicks and keystrokes to determine when to fire a click event. What this means is adding two new Windows message handlers to our code; one monitors the left mouse button, whereas the other monitors the Enter key. The following procedure shows you how to add a Windows message handler to the example.

Try This

1. Right-click CMultPB2Ctrl in the ClassView tab of the Workspace window; then choose Add Windows Message Handler from the context menu. You should see the New Windows Message and Event Handlers dialog box shown in Figure 13-35. This dialog box lists all of the handlers for the current class (as specified in the dialog box title bar).

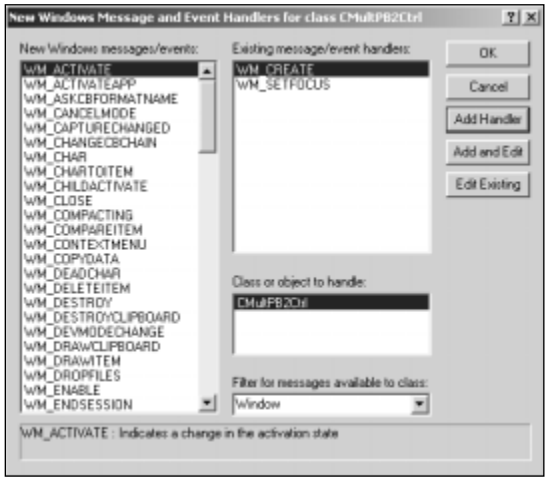


Figure 13-35: The New Windows Message and Event Handlers dialog box allows you to add event and message handlers to various parts of your control.

2. Choose Child Window in the Filter for messages available to class field. We need to monitor the pushbutton events. Since the pushbutton is actually within a container, we need to monitor the child window, not the window itself. This changes the listing of available messages to those specifically sent to the child window.
3. Highlight the WM_LBUTTONDOWN message; then click Add Handler. This adds a new handler for a left mouse button down event.

4. Highlight the WM_KEYDOWN, then click Add and Edit. Visual C++ adds the new message handler and takes you directly to the new functions.

Now that we have some new functions to work with, let's look at the code required to make them work. Listing 13-14 shows the code for the WM_LBUTTONDOWN and WM_KEYDOWN message handlers. Normally, these are the only two handlers that you require for a pushbutton.

Listing 13-14: OnKeyDown() and OnLButtonDown() Message Handler Source Code

```
LRESULT OnKeyDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
  // Perform the normal button processing.
  m_ctlButton.DefWindowProc(uMsg, wParam, lParam);
  // Determine the current key code.  If the key code is
  // the Enter key, then change the button status; otherwise,
  // exit.
  if (!(wParam == 13))
    return 0;
  // Determine what the next button position is.
  if (iCurrentChoice < iDisplaySize - 1)
  {
    iCurrentChoice++;
    m_modalResult++;
  }
  else
  {
    m_modalResult = m_modalResult - iCurrentChoice;
    iCurrentChoice = 0;
  }

  // Display the button text for the current position.
  if (iDisplaySize > 1)
    m_ctlButton.SetWindowText(aDisplayValues[iCurrentChoice]);
  // Perform the default action.
  Fire_Click();

  return 0;
}

LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
  // Perform the normal button processing.
  m_ctlButton.DefWindowProc(uMsg, wParam, lParam);
  // Determine what the next button position is.
```

Continued

Listing 13-14: *(continued)*

```
  if (iCurrentChoice < iDisplaySize - 1)
  {
    iCurrentChoice++;
    m_modalResult++;
  }
  else
  {
    m_modalResult = m_modalResult - iCurrentChoice;
    iCurrentChoice = 0;
  }

  // Display the button text for the current position.
  if (iDisplaySize > 1)
    m_ctlButton.SetWindowText(aDisplayValues[iCurrentChoice]);
  // Perform the default action.
  Fire_Click();

  return 0;
}
```

Both of these methods do the same thing — they allow the control to detect user input. The OnKeyDown( ) method begins by performing one check that the OnLButtonDown( ) method doesn't need to perform. It uses the wParam input to detect which key the user pressed. The OnKeyDown( ) method reacts only to the Enter key. Both methods allow the default procedure to perform whatever tasks it needs to do first by calling the DefWindowProc( ) method of the m_ctlButton object. (This object is created automatically for you by the ATL wizard and allows the control to handle standard button events.) The button-changing process works the same as the MFC version of the control. Even though the method used to fire a click differs, this process works the same as the MFC version of the control as well.

Adding Some Initialization Code

Initialization for the ATL version of our control is much more complicated than the MFC version. For one thing, ATL assumes nothing about the initial state of the variables and, unlike MFC, there isn't any way to define an initial state as part of the process of making the properties persistent. Obviously, we need another initialization alternative, which is to add both the Load( ) and InitNew( ) methods to the control. The InitNew( ) method gets called to initialize the control's properties if you create a new control, while the Load( ) method gets called to load the properties for an existing control. Listing 13-15 shows the source code for both the Load( ) and InitNew( ) methods.

Listing 13-15: **Load() and InitNew() Method Source Code**

```
STDMETHODIMP CMultPB2Ctrl::InitNew()
{
  CComPtr<IFont>  p;    // The ambient (default) font pointer.
  CComPtr<IFont>  pFont;  // A copy of the font for local use.
  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Get the default font, create a copy of it, then initialize // our Font
property with it.  if(!m_pFont)
    if(SUCCEEDED(GetAmbientFont(&p)) && p)
      if(SUCCEEDED(p->Clone(&pFont)) && pFont)
        pFont->QueryInterface(IID_IFontDisp, (void**)&m_pFont);
  // Initialize the rest of the properties.
  m_bEnabled = true;
  m_bstrCaption = T2BSTR("On");
  m_buttonType = 0;
  m_modalResult = 1;

  // Set the initial control size.
  m_sizeExtent.cx = 2100;
  m_sizeExtent.cy = 580;

  // Register the update
  SetDirty(true);
  FireViewChange();

  // Set the initialization variable so that we don't
  // initialize the control twice.
  m_bInitialized = true;
  // Return a non-error result.
  return S_OK;
}

STDMETHODIMP CMultPB2Ctrl::Load(LPSTREAM pStream)
{
  // If the control has already been initialized by InitNew(),
  // the return E_UNEXPECTED so that the client knows that the
  // control has already been initialized.
  if (m_bInitialized)
    return E_UNEXPECTED;
  else
  {
    // Otherwise, set m_bInitialized to true and perform
    // the default initialization.
    m_bInitialized = true;
    return IPersistStreamInitImpl<CMultPB2Ctrl>::Load(pStream);
  }
}
```

Let's talk about the InitNew( ) method first. This method gets called when creating a new control. Since this is a new control, there's a good chance that some properties aren't defined at all. So, the first bit of code checks to ensure the Font property is defined. If not, the method assigns the current system font to the control. Initializing the other properties is as easy as assigning them a value. You want to be sure that all of the defaults you assign work together. In this case, the button starts out as an On/Off button in the default On state. All ATL controls start out as moderately large squares, which isn't a very good shape for a pushbutton. The next bit of code assigns a more believable size to the control that roughly equates to the size of a standard Visual C++ pushbutton. Once all the initialization is complete, you must use the SetDirty( ) method to ensure the control values get changed, and the FireViewChange( ) method to ensure the changes also get reflected in the control's appearance. Finally, the InitNew( ) method tells the control that it's initialized by setting m_bInitialized to true, then returns a nonerror result to the control.

The Load( ) method always gets called — even for a new control. That's why the m_bInitialized variable is so important. If the control has been initialized by InitNew( ), the Load( ) method has to return a value of E_UNEXPECTED. Otherwise, the control could get initialized twice, resulting in unexpected results.

If the control is initialized, then it's safe for the Load( ) method to load the persisted properties for the control. In this case, we simply need to call on the Load( ) method of the IPersistStreamInit interface. IPersistStreamInit also includes methods for saving and initializing a new control. In fact, the InitNew( ) method in this example overrides the IPersistStreamInit interface implementation of the same method. Depending on the situation, you may need to use one or more of these other methods in your control. In this particular case, all we need to do is override the default Load( ) implementation to check on the control's initialization status. Since the default InitNew( ) method doesn't do anything in this case, we don't need to include it as part of the InitNew( ) method.

Another potential problem is the appearance of the control itself. You may not see the control you expected to see when you place in an IDE. Part of the problem is that ATL makes no assumptions about how you want to create the control — it creates the most basic form available without taking any special features of a subclassed control into account. With that in mind, you need to change the control's OnCreate( ) method as shown in Listing 13-16 to take better advantage of the inherent features that control has to offer.

Listing 13-16: **OnCreate( ) Method Source Code**

```
  LRESULT OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/, BOOL&
/*bHandled*/)
  {
    // Determine how big to make the control.
    // Tell Visual C++ that we'll be converting BSTR values.
    USES_CONVERSION;
```

```
    // Determine how big to make the control.
    RECT rc;
    GetWindowRect(&rc);
    rc.right -= rc.left;
    rc.bottom -= rc.top;
    rc.top = rc.left = 0;

    // Make sure you modify the default Create() method syntax
    // to create the kind of button that you want.
    m_ctlButton.Create(m_hWnd, rc,
      OLE2T(m_bstrCaption),
      BS_PUSHBUTTON | WS_VISIBLE | WS_CHILD | WS_TABSTOP);
    // Return a non-error result.
    return 0;
  }
```

For the most part, you'll find that ATL creates a simple box for you to look at, and it's up to you to do something with it. In this case, we add a caption to the pushbutton and give it some of the standard pushbutton flags. The flags you add depend on the type of control you create. In some cases, the default ATL settings work just fine, but in most cases, you need to make some minor changes. Obviously, the flags you choose to use have to be supported by the control you create.

Visual C++ users will see a problem almost instantly after they compile the control code and try to use it within a Visual C++ application. What you get for a default presentation in the IDE is a blank white box. Fortunately, the control appears as normal when you compile and run the application. The problem is that Visual C++ doesn't InPlaceActivate the control like the IDEs for other language products such as Visual Basic. As a result, the OnPaint( ) method never gets called to paint the control onscreen, and the ATL implementation for OnDraw( ) doesn't do anything. Listing 13-17 shows one way to get around this problem by implementing the OnDraw( ) method yourself.

Listing 13-17: **OnDraw( ) Method Source Code**

```
HRESULT CMultPB2Ctrl::OnDraw(ATL_DRAWINFO &di)
{
  // Control's bounding rectangle.
  RECT& rc = *(RECT*)di.prcBounds;

  // If the control isn't InPlaceActive, then we
```

Continued

Listing 13-17: *(continued)*

```
  // need to draw it.  Otherwise, the control gets
  // drawn as part of a call to OnPaint().
  if (!m_bInPlaceActive)
  {
    // Tell Visual C++ that we'll be converting BSTR values.
    USES_CONVERSION;
    // Make the control display a simple rectangle with text.
    Rectangle(di.hdcDraw,
      rc.left,
      rc.top,
      rc.right,
      rc.bottom);
    DrawText(di.hdcDraw,
      OLE2T(m_bstrCaption),
      -1, &rc, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
  }

  return S_OK;
}
```

I chose to create a very simple presentation, in this case, because the drawing code isn't all that important. All the user really needs is a box to show the extent of the control and the caption text. Creating an image that looks like a button is nice but not absolutely required.

There's one absolute necessity. Notice that the code checks the InPlaceActivate status of the control before it draws anything. You must perform this step or the button appearance will be flawed when the application is running.

Adding the Font and Custom Property Pages

In the MFC example, most of the work required to attach the custom property page to the control is done for you. All you really need to do is place some controls on the property page dialog box, then create member variables that attach themselves to the OLE property associated with the control. When working with ATL, you can't create member variables for the controls on the property sheet using this method — you need to create everything manually.

The first task you need to perform after adding controls to the property page dialog box is manually add the same four member variables (m_bstrCaption, m_bEnabled, m_modalResult, and m_buttonType) that we used for the control portion of the example. Use the same method we used to add member variables to the CMultPB2PropPage entry in the ClassView tab of the Workspace window.

Once you have some member variables in place, you need some method to get the current control values and place them in the dialog box. That's the purpose of the OnInitDialog( ) method, which you add using the WM_INITDIALOG Windows message handler. You use the same technique to add this message handler that we used to add the WM_LBUTTONDOWN and WM_KEYDOWN message handlers to the control. Listing 13-18 shows the source code for the OnInitDialog( ) method.

Listing 13-18: **OnInitDialog( ) Method Source Code**

```
LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
  UINT    iCount;             // Counter.
  LPTSTR  lpszModalResult;    // String version of ModalResult.

  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Create a pointer to the control class.
  IMultPB2Ctrl*   pMultPB2Ctrl;

  // Keep querying interfaces until finished.
  for (iCount = 0; iCount < m_nObjects; iCount++)
  {

    // Query the interface.
    m_ppUnk[iCount]->QueryInterface(IID_IMultPB2Ctrl,
                                    (void**)&pMultPB2Ctrl);
    // Get the current property values.
    pMultPB2Ctrl->get_ModalResult(&m_modalResult);
    pMultPB2Ctrl->get_ButtonType(&m_buttonType);
    pMultPB2Ctrl->get_Enabled(&m_bEnabled);
    pMultPB2Ctrl->get_Caption(&m_bstrCaption);

    // Display the ModalResult value.
    lpszModalResult = "0";
    itoa(m_modalResult, lpszModalResult, 10);
    SetDlgItemText(IDC_MODALRESULT, lpszModalResult);

    // Set the current button type.
    switch(m_buttonType)
    {
    case 0:
      CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_ON_OFF);
      break;
    case 1:
      CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_YES_NO);
      break;
```

Continued

Listing 13-18: *(continued)*

```
    case 2:
      CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_RED_YELLOW_GREEN);
      break;
    case 3:
      CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_USER_MODE);
      break;
    }

    // Set the Enabled state.
    CheckDlgButton(IDC_ENABLED, m_bEnabled);

    // Set the Caption field.
    SetDlgItemText(IDC_CAPTION, OLE2T(m_bstrCaption));
    // Release the interface.
    pMultPB2Ctrl->Release();
  }
  return 0;
}
```

The property page and control aren't connected in any way. In fact, if you look at any ActiveX control with a utility like OLE View, you can see that each property page has its own *GUID (globally unique identifier)* that's completely separate from the control. So the first thing we need to do is create a connection to the control by defining and initializing an interface pointer. Querying the interface allows us to fill local variables with the current values of each control property, then display those values on the property page dialog box.

Of course, we can't display these property values directly in most cases. Note that since the ModalResult property is a number, we need to convert it to a text value before displaying it. The ButtonType and Enabled properties aren't displayed directly. The value they contain is used to modify the state of the property page buttons. Even the Caption property, which is a text value, requires conversion before we can display it since this property is a BSTR.

The very last thing we do in the InitDialog( ) method is release the interface. You must do this or suffer unpleasant consequences. At the very least, your control will produce a memory leak and prevent Windows from releasing the control when all clients are finished with it. One of the main purposes of the Release( ) method is to ensure the control maintains an accurate reference count. When the reference count is zero, the control can destroy itself and free memory for other purposes.

Modifying the property page values comes next. There are four ButtonType related buttons, an Enabled button, and a Caption edit box, all of which require an

individual handler. You can add handlers for each of the buttons by adding a BN_CLICKED message handler. The Caption edit box requires an EN_CHANGED message handler. Listing 13-19 shows the code for these message handlers.

Listing 13-19: **Property Page Control Message Handlers**

```
LRESULT OnChangeCaption(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
{
  LPTSTR  lpszTitle = " ";    // New caption value.
  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Get the current edit box text.
  GetDlgItemText(IDC_CAPTION, lpszTitle, 40);

  // Set the properties.
  m_bstrCaption = T2BSTR(lpszTitle);

  // Enable the Apply button for the Property Page dialog.
  IPropertyPageImpl<CMultPB2PropPage>::SetDirty(TRUE);
  return 0;
}

LRESULT OnClickedEnabled(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
{
  // Reverse the current Enabled value.
  m_bEnabled = !m_bEnabled;

  // Enable the Apply button for the Property Page dialog.
  IPropertyPageImpl<CMultPB2PropPage>::SetDirty(TRUE);
  return 0;
}

LRESULT OnClickedOn_off(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
{
  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Set the properties.
  m_bstrCaption = T2BSTR("On");
  m_buttonType = 0;
  m_modalResult = 1;

  // Enable the Apply button for the Property Page dialog.
  IPropertyPageImpl<CMultPB2PropPage>::SetDirty(TRUE);
```

Continued

Listing 13-19: *(continued)*

```
  return 0;
}

LRESULT OnClickedRed_yellow_green(WORD wNotifyCode, WORD wID, HWND hWndCtl,
BOOL& bHandled)
{
  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Set the properties.
  m_bstrCaption = T2BSTR("Red");
  m_buttonType = 2;
  m_modalResult = 5;

  // Enable the Apply button for the Property Page dialog.
  IPropertyPageImpl<CMultPB2PropPage>::SetDirty(TRUE);
  return 0;
}

LRESULT OnClickedUser_mode(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
{
  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Set the properties.
  m_bstrCaption = T2BSTR("Button");
  m_buttonType = 3;
  m_modalResult = -1;

  // Enable the Apply button for the Property Page dialog.
  IPropertyPageImpl<CMultPB2PropPage>::SetDirty(TRUE);
  return 0;
}

LRESULT OnClickedYes_no(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
{
  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Set the properties.
  m_bstrCaption = T2BSTR("Yes");
  m_buttonType = 1;
  m_modalResult = 3;

  // Enable the Apply button for the Property Page dialog.
  IPropertyPageImpl<CMultPB2PropPage>::SetDirty(TRUE);
  return 0;
}
```

None of these functions are very complicated. For the most part, they simply collect the values required from the property page and place them in the memory variables associated with that property. Of course, the problem now is to provide some type of signal that the property values have changed. That's what the SetDirty( ) method is all about. It signals the control that the property page values have changed. One of the side effects of this method is that the Apply button on the property page gets activated, telling the user that the values have changed and that one last step is required to accept them.

At this point, you have a method for getting the current property values and a series of methods for accepting new ones. The last task is to create a method that applies the new values to the control once users are sure of what values they want to use. This particular method is provided as part of the initial property page code — all you need to do is enhance. Listing 13-20 shows the code you need to add to the Apply( ) method.

Listing 13-20: **Apply( ) Method Source Code**

```
STDMETHOD(Apply)(void)
{
  UINT    iCount;             // Counter.
  LPTSTR  lpszModalResult;    // String ModalResult.

  // Tell Visual C++ that we'll be converting BSTR values.
  USES_CONVERSION;
  // Create a pointer to the control class.
  IMultPB2Ctrl*   pMultPB2Ctrl;

  // Keep querying interfaces until finished.
  for (iCount = 0; iCount < m_nObjects; iCount++)
  {
    // Query the interface.
    m_ppUnk[iCount]->QueryInterface(IID_IMultPB2Ctrl, (void**)&pMultPB2Ctrl);

    // Apply the new property values.
    pMultPB2Ctrl->put_ButtonType(m_buttonType);
    pMultPB2Ctrl->put_Enabled(m_bEnabled);
    pMultPB2Ctrl->put_Caption(m_bstrCaption);

    // Release the interface.
    pMultPB2Ctrl->Release();
  }

  // Display the ModalResult value.
  lpszModalResult = "0";
```

Continued

Listing 13-20: *(continued)*

```
  itoa(m_modalResult, lpszModalResult, 10);
  SetDlgItemText(IDC_MODALRESULT, lpszModalResult);

  // Set the Caption field.
  SetDlgItemText(IDC_CAPTION, OLE2T(m_bstrCaption));
  // Tell the control that the property page values // have changed.  m_bDirty =
FALSE;
  // Return a non-error result.
  return S_OK;
}
```

The Apply( ) method follows a similar, though somewhat different, sequence of events as the InitDialog( ) method. The code still needs to acquire a pointer to the control. Once the control gets the interface pointer, it can move the values from the property page memory variables to the memory variables within the control. Once the method has moved the property values to the control, it can set m_bDirty to FALSE, indicating that the property values are no longer different from those in the control.

If you work with a lot of ATL controls that use fonts, it's always good to have a utility around that can actually view the fonts that you want to use. Normally, you can use the Character Map utility that comes with Windows to view the font and even select special characters as needed. Unfortunately, this utility doesn't work well with Unicode fonts. The solution for this problem can be found on Heiner Eichmann's Web site at `http://sun1.rrzn.uni-hannover.de/nhbieich/listfont.htm`. The ListFont utility will allow you to see everything that a font has to offer. It also provides a complete enumeration of all of the fonts on your machine, some of which may be hidden in the Character Map utility.

Designing Listboxes for the ModalResult and ButtonType Properties

Remember from the MFC control discussion that adding listboxes containing acceptable property values to your control means implementing the IPerPropertyBrowsing interface. Adding a new interface to an ATL control requires a little more work than with MFC — but only a little more.

In this case, we need to add two entries to the MultPB2Ctrl.H file. The first tells Visual C++ that we want to derive a control that includes a predefined IPerPropertyBrowsing implementation. The second allows the client to see the interface as implemented. You need to perform these two steps even if you don't intend to override any of the default method implementations within the interface. Listing 13-21 shows the entry you need to add to derive the control using the IPerPropertyBrowsing interface in bold. Listing 13-22 shows the entry that you need to add to make the interface visible to a client.

Listing 13-21: **Deriving a Control with an Additional Interface**

```
class ATL_NO_VTABLE CMultPB2Ctrl :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CStockPropImpl<CMultPB2Ctrl, IMultPB2Ctrl, &IID_IMultPB2Ctrl,
&LIBID_MULTPB2Lib>,
   public CComControl<CMultPB2Ctrl>,
   public IPersistStreamInitImpl<CMultPB2Ctrl>,
   public IOleControlImpl<CMultPB2Ctrl>,
   public IOleObjectImpl<CMultPB2Ctrl>,
   public IOleInPlaceActiveObjectImpl<CMultPB2Ctrl>,
   public IViewObjectExImpl<CMultPB2Ctrl>,
   public IOleInPlaceObjectWindowlessImpl<CMultPB2Ctrl>,
   public IConnectionPointContainerImpl<CMultPB2Ctrl>,
   public IPersistStorageImpl<CMultPB2Ctrl>,
   public ISpecifyPropertyPagesImpl<CMultPB2Ctrl>,
   public IQuickActivateImpl<CMultPB2Ctrl>,
   public IDataObjectImpl<CMultPB2Ctrl>,
   public IProvideClassInfo2Impl<&CLSID_MultPB2Ctrl, &DIID__IMultPB2CtrlEvents,
&LIBID_MULTPB2Lib>,
   public IPropertyNotifySinkCP<CMultPB2Ctrl>,
   public CComCoClass<CMultPB2Ctrl, &CLSID_MultPB2Ctrl>,
   public CProxy_IMultPB2CtrlEvents< CMultPB2Ctrl >,

   // Add the property browing implemenation class.
   public IPerPropertyBrowsingImpl<CMultPB2Ctrl>
```

Listing 13-22: **Making the Interface Visible to the Client**

```
BEGIN_COM_MAP(CMultPB2Ctrl)
   COM_INTERFACE_ENTRY(IMultPB2Ctrl)
   COM_INTERFACE_ENTRY(IDispatch)
   COM_INTERFACE_ENTRY(IViewObjectEx)
   COM_INTERFACE_ENTRY(IViewObject2)
   COM_INTERFACE_ENTRY(IViewObject)
   COM_INTERFACE_ENTRY(IOleInPlaceObjectWindowless)
   COM_INTERFACE_ENTRY(IOleInPlaceObject)
   COM_INTERFACE_ENTRY2(IOleWindow, IOleInPlaceObjectWindowless)
   COM_INTERFACE_ENTRY(IOleInPlaceActiveObject)
   COM_INTERFACE_ENTRY(IOleControl)
   COM_INTERFACE_ENTRY(IOleObject)
   COM_INTERFACE_ENTRY(IPersistStreamInit)
   COM_INTERFACE_ENTRY2(IPersist, IPersistStreamInit)
   COM_INTERFACE_ENTRY(IConnectionPointContainer)
   COM_INTERFACE_ENTRY(ISpecifyPropertyPages)
```

Continued

Listing 13-22: *(continued)*

```
   COM_INTERFACE_ENTRY(IQuickActivate)
   COM_INTERFACE_ENTRY(IPersistStorage)
   COM_INTERFACE_ENTRY(IDataObject)
   COM_INTERFACE_ENTRY(IProvideClassInfo)
   COM_INTERFACE_ENTRY(IProvideClassInfo2)
   COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)

   // Add the property browsing interface.
   COM_INTERFACE_ENTRY(IPerPropertyBrowsing)
END_COM_MAP()
```

As you can see, the entries we need to make are simple. Every ATL control you work with should include a series of interface definitions like these by default. If you find that you need to add some functionality to the control by adding a new interface, then it's an easy matter to do so. Likewise, removing an unneeded interface is as easy as removing the required control derivation and COM map entries, along with any methods that you override for the interface.

Tip ATL provides a lot of default interface implementations for you. Simply look for the name of the interface with "Impl" added to the end. For example, this control requires the IPerPropertyBrowsing interface that actually requires four methods. Using the IPerPropertyBrowsingImpl template allows us to add the default implementation of this interface to the control, then override only the methods that require change.

Once you have the two interface entries in place, you need to override the three same methods that we did for the MFC version of the control. Simply add the GetDisplayString( ), GetPredefinedStrings( ), and GetPredefinedValue( ) methods as before. Add them as simple member functions with a return value of STDMETHODIMP. Listing 13-23 shows the code you need to add to implement these methods.

Listing 13-23: **IPerPropertyBrowsing Implementation Source Code**

```
STDMETHODIMP CMultPB2Ctrl::GetDisplayString(DISPID dispID, BSTR *pBstr)
{
   // Tell Visual C++ that we'll be converting BSTR values.
   USES_CONVERSION;
   // Determine if we need to display a property // value string.  switch
(dispID)
```

```
{

// Display a ButtonType property value.
case DISPID_BUTTONTYPE:

   // Determine which value to display.
   switch (m_buttonType)
   {
   case 0:
      *pBstr = SysAllocString(OLESTR("0 - On/Off"));
      break;
   case 1:
      *pBstr = SysAllocString(OLESTR("1 - Yes/No"));
      break;
   case 2:
      *pBstr = SysAllocString(OLESTR("2 - Red/Yellow/Green"));
      break;
   case 3:
      *pBstr = SysAllocString(OLESTR("3 - User Mode"));
      break;
   }

   // Return true so the programming environment
   // knows we returned a value.
   return S_OK;
// Display a ModalResult property value.
case DISPID_MODALRESULT:

   // Determine which value to display.
   switch (m_modalResult)
   {
   case -1:
      *pBstr = SysAllocString(OLESTR("-1 - User Mode"));
      break;
   case 1:
      *pBstr = SysAllocString(OLESTR("1 - On"));
      break;
   case 2:
      *pBstr = SysAllocString(OLESTR("2 - Off"));
      break;
   case 3:
      *pBstr = SysAllocString(OLESTR("3 - Yes"));
      break;
   case 4:
      *pBstr = SysAllocString(OLESTR("4 - No"));
      break;
   case 5:
      *pBstr = SysAllocString(OLESTR("5 - Red"));
      break;
```

Continued

Listing 13-23: *(continued)*

```
      case 6:
         *pBstr = SysAllocString(OLESTR("6 - Yellow"));
         break;
      case 7:
         *pBstr = SysAllocString(OLESTR("7 - Green"));
         break;
      }

      // Return true so the programming environment
      // knows we returned a value.
      return S_OK;
   }

   // Perform the default action.
   return IPerPropertyBrowsingImpl<CMultPB2Ctrl>::GetDisplayString(dispID,
pBstr);
}

STDMETHODIMP CMultPB2Ctrl::GetPredefinedStrings(DISPID dispID, CALPOLESTR
*pCaStringsOut, CADWORD *pCaCookiesOut)
{
   ULONG iCount;  // Loop counter.
   // Set the ButtonType string values.
   static const LPCOLESTR  aBTStrings[] =
      {
         OLESTR("0 - On/Off"),
         OLESTR("1 - Yes/No"),
         OLESTR("2 - Red/Yellow/Green"),
         OLESTR("3 - User Mode")
      };

   // Set the ButtonType cookie values.
   static const DWORD   aBTCookies[] = {0, 1, 2, 3};
   // Set the ModalResult string values.
   static const LPCOLESTR  aMRStrings[] =
      {
         OLESTR("-1 - User Mode"),
         OLESTR("1 - On"),
         OLESTR("2 - Off"),
         OLESTR("3 - Yes"),
         OLESTR("4 - No"),
         OLESTR("5 - Red"),
         OLESTR("6 - Yellow"),
         OLESTR("7 - Green")
      };

      // Set the ModalResult cookie values.
   static const DWORD   aMRCookies[] =
```

```
   {-1, 1, 2, 3, 4, 5, 6, 7};
// Verify that there are structures to use.
if (NULL == pCaStringsOut || NULL == pCaCookiesOut)
   return E_POINTER;
// Clear the structure entries.
pCaStringsOut->cElems = 0;
pCaStringsOut->pElems = NULL;
pCaCookiesOut->cElems = 0;
pCaCookiesOut->pElems = NULL;

// Determine which property to process.
switch (dispID)
{
case DISPID_BUTTONTYPE:
   // Set the number of ButtonType strings.
   pCaStringsOut->cElems = 4;

   // Allocate an array of string pointers
   pCaStringsOut->pElems = (LPOLESTR*)
      CoTaskMemAlloc (4 * sizeof(LPOLESTR));
   if (NULL == pCaStringsOut->pElems)
      return E_OUTOFMEMORY;
   // Fill the array with strings.
   for (iCount = 0; iCount < 4; iCount++)
   {
     // Allocate memory for the string and terminating NULL character
     pCaStringsOut->pElems[iCount] = (LPOLESTR)
        CoTaskMemAlloc ((ocslen(aBTStrings[iCount]) + 1) *
        sizeof(OLECHAR));
     if (NULL == pCaStringsOut->pElems[iCount])
        return E_OUTOFMEMORY;
     // Add the string to the array.
     ocscpy(pCaStringsOut->pElems[iCount], aBTStrings[iCount]);
   }

   // Set the number of ButtonType cookies.
   pCaCookiesOut->cElems = 4;

   // Allocate array of cookies
   pCaCookiesOut->pElems = (LPDWORD)
      CoTaskMemAlloc (4 * sizeof(DWORD));
   if (NULL == pCaCookiesOut->pElems)
      return E_OUTOFMEMORY;
   // Fill the array with cookies.
   ::CopyMemory (pCaCookiesOut->pElems,
      aBTCookies,
      4 * sizeof (DWORD));

   return S_OK;
```

Continued

Listing 13-23: *(continued)*

```
   case DISPID_MODALRESULT:
      // Set the number of ModalResult strings.
      pCaStringsOut->cElems = 8;
      // Allocate an array of string pointers
      pCaStringsOut->pElems = (LPOLESTR*)
         CoTaskMemAlloc (8 * sizeof(LPOLESTR));
      if (NULL == pCaStringsOut->pElems)
         return E_OUTOFMEMORY;
      // Fill the array with strings.
      for (iCount = 0; iCount < 8; iCount++)
      {
        // Allocate memory for the string and terminating NUL character
        pCaStringsOut->pElems[iCount] = (LPOLESTR)
           CoTaskMemAlloc ((ocslen(aMRStrings[iCount]) + 1) *
           sizeof(OLECHAR));
        if (NULL == pCaStringsOut->pElems[iCount])
           return E_OUTOFMEMORY;
        // Add the string to the array.
        ocscpy(pCaStringsOut->pElems[iCount], aMRStrings[iCount]);
      }

      // Set the number of ModalResult cookies.
      pCaCookiesOut->cElems = 8;
      // Allocate array of cookies
      pCaCookiesOut->pElems = (LPDWORD)
         CoTaskMemAlloc (8 * sizeof(DWORD));
      if (NULL == pCaCookiesOut->pElems)
         return E_OUTOFMEMORY;
      // Fill the array with cookies.
      ::CopyMemory (pCaCookiesOut->pElems,
         aMRCookies,
         8 * sizeof (DWORD));

      return S_OK;
   }

   // Perform the default action.
   return IPerPropertyBrowsingImpl<CMultPB2Ctrl>::GetPredefinedStrings(dispID,
pCaStringsOut, pCaCookiesOut);
}

STDMETHODIMP CMultPB2Ctrl::GetPredefinedValue(DISPID dispID, DWORD dwCookie,
VARIANT *pVarOut)
{
   // Determine which property to react to.
   switch (dispID)
   {
```

```
   // Return a value for the predefined ButtonType string.
   case DISPID_BUTTONTYPE:
      // Initialize the variant structure.
      VariantInit(pVarOut);

      // Add the type and value of the output.
      pVarOut->vt = VT_I2;
      pVarOut->iVal = short(dwCookie);

      // Return true to indicate we've set the values.
      return S_OK;
   // Return a value for the predefined ButtonType string.
   case DISPID_MODALRESULT:
      // Initialize the variant structure.
      VariantInit(pVarOut);

      // Add the type and value of the output.
      pVarOut->vt = VT_I2;
      pVarOut->iVal = short(dwCookie);

      // Return true to indicate we've set the values.
      return S_OK;
   }

   // Perform the default action.
   return IPerPropertyBrowsingImpl<CMultPB2Ctrl>::GetPredefinedValue(dispID,
dwCookie, pVarOut);
}
```

Although the method names are the same as the MFC version, at least two of the methods are completely different. The only method that looks similar in both versions is the GetPredefinedValue( ) method. Interestingly enough, the MFC and ATL versions of this method both provide a DWORD value (cookie) as input, and require essentially the same output. You can move the GetPredefinedValue( ) method code from MFC to ATL with very few changes. However, this is the exception rather than the rule, so always be careful when making assumptions about your MFC code in an ATL environment.

Notice that the code does perform the default GetDisplayString( ) action once the custom implementation is completed. Though MFC also performs this task, the method you need to use to perform it in ATL differs somewhat. Make sure you call the interface implementation, not a class such as COleControl when using ATL.

The calling syntax for both GetPredefinedStrings( ) and GetDisplayString( ) is different in ATL than it is in MFC. This difference is important because you can't use the same code in both types of controls. Make certain that you use the interface version of the method calling syntax when developing an ATL control. In most

cases, you may find that MFC and ATL are completely incompatible when it comes to this particular area of control development. In other words, don't assume you can simply move your MFC code to ATL to derive the benefits that ATL provides to the programmer.

The GetDisplayString( ) method is the simplest of the remaining two methods. That's because this method only returns a string that describes the current property value. Needless to say, returning a string is the same whether you use ATL or MFC (though the actual format of the string varies). In this case, let's use the SysAllocString( ) function to create the BSTR output required for the control. Notice that there are actually two conversion steps in this method. First, we convert the LPTSTR to an OLESTR before allocating the memory. The string is converted to a BSTR as part of the memory allocation process.

The GetPredefinedStrings( ) method begins by creating arrays for each property that you want to support. There are two arrays for each property. The first contains the string that gets displayed when the user opens the listbox. The second array contains a corresponding list of cookie values that get passed to GetPredefinedValue( ) when the user makes a selection. The cookie values are always DWORD values and don't necessarily have to match the values of the property. Both of these arrays have to be the same length.

The next thing the code has to do is check the input variables. These variables contain NULL in some cases rather than the structures that you need to fill for the programming language IDE. This is especially true with Visual Basic where the IDE always calls this method during control initialization with a NULL rather than a blank structure. Make sure you return an error value of E_POINTER when this happens so that the IDE passes your method the blank structures required to display property values for the control.

Clearing the structure entries comes next. If you don't perform this step, the control may display false values for a certain property. There's no guarantee that the programming language IDE will actually pass blank structures—just that the structures will contain the required number of elements.

Filling the structures comes next. Make sure you allocate memory using CoTaskMemAlloc( ) rather than malloc( ). The CoTaskMemAlloc( ) method allows the client to free the memory later. Notice that you also need to add extra memory for the terminating NULL characters of strings. Otherwise, the strings run into each other when displayed on the client or a memory error results. Finally, allocating memory for the array of strings is a two-step process. First, you need to allocate memory for the pointers to the string values, then allocate memory for the strings themselves.

What's Missing?

At this point, we've implemented all of the features defined in the specification for the Multiple State Pushbutton control. If you use the ATL and MFC version of this

control within the same application, the end user probably won't notice any difference between the two. However, there are still some things missing from the ATL version of the control and you need to be aware of them.

The most obvious missing elements are full implementations of the three stock properties for this control. That's right, if you try using them right now, they do only part of the work they're designed to do. The MFC version of the control provides stock properties that are fully functional without any added coding. The same can't be said for the ATL version of the control. The number one rule you must remember when working with ATL is that nothing is included. ATL makes no assumptions when it comes to what you want to put into the control, which is simultaneously a source of great flexibility and a lot of extra work for the developer.

Once you get past the property implementations, there's another comparison you need to make. Interfaces are the bread and butter of component technology. If you want to create two controls that are exactly the same, then you need to make sure these controls provide exactly the same interfaces. Figure 13-36 shows the interfaces for the MFC and ATL versions of the Multiple State Pushbutton.

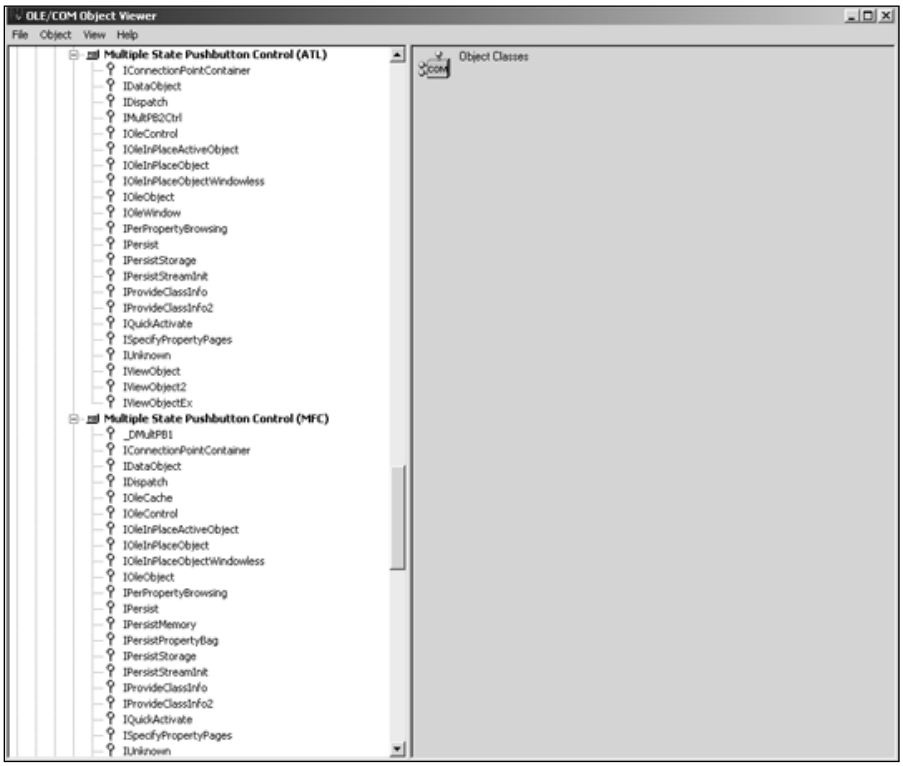


Figure 13-36: The MFC and ATL versions of the Multiple State Pushbutton don't provide exactly the same interfaces.

A simple survey shows that the ATL version of the Multiple State Pushbutton lacks three interfaces that the MFC version provides: IOleCache, IPersistMemory, and IPersistPropertyBag. The MFC version also lacks one of the interfaces that the ATL version provides: IOleWindow. The following list describes each of these missing interfaces.

IOleCache	This interface determines what data gets cached inside a control for the container's use. In most cases, this cached data includes presentation information so that the container can display the control without activating it. The lack of this interface is the main reason we need to provide an OnDraw( ) method for the ATL version of the control. However, implementing OnDraw( ) is still easier than implementing the entire IOleCache interface. You'll need to decide how valuable this interface will be before you implement it.
IOleWindow	This interface allows the container to obtain a handle to each of the windows used for in-place activation. It also allows the user to enter and exit context-sensitive help mode. You can gain the functionality of IOleWindow by deriving other interfaces from it. In this case, the MFC version of the control does gain this functionality through interfaces such as the IOleInPlaceObject interface.
IPersistMemory	This interface behaves exactly the same way as the IPersistStreamInit interface, which is implemented by both versions of the control. The main difference between the two interfaces is that IPersistMemory allows the container to use a fixed-size memory block.
IPersistPropertyBag	This interface allows the container to store the values contained in individual properties. You use it with the IPropertyBag and IErrorLog to allow a container to store and retrieve individual properties. The alternative is to use an interface such as IPersistStream, which stores all of the properties at one time in a single data stream. (The IPersistStream interface is implemented by the container — the control uses IPersistStreamInit.) In most cases, you may find that IPersistPropertyBag is far more flexible than using streams to save your persistence data.

Creating Components the COM+ Way

Right now, there are only two completely documented ways to create components using Visual C++. You can go the MFC route, which allows fast development of very robust controls, or you can go the ALT route, which takes longer, but offers greater flexibility. Soon, there will be a third route that you can take when creating components that rely on COM+. Unfortunately, the pre-alpha COM+ SDK available at the time of this writing wouldn't allow development of a third version of the Multiple State Pushbutton control.

So what will a COM+ component look like? From a coding perspective, it should require less coding than an ATL control, but still more than an MFC control. You'll implement a COM+ component using techniques similar to those used to create standard classes right now. In other words, COM+ programming will require little more expertise than the ability to create a class. You'll still have to provide interface information, but a special compiler will take care of many of the interface description details that you normally need to take care of manually.

As you can see, the OLE/COM Object Viewer provides you with a lot of information about the controls installed on your system and can help you troubleshoot problems. You can also use it to compare two controls to see why one works in a specific environment, whereas another doesn't. In fact, you can even use this technique to research a control that you're building. For example, you may want to make your control scriptable, but have no idea how to do so. Looking through the list of interfaces implemented by other scriptable controls provides you with some ideas on how to make your control scriptable as well.

A Comparison of the Results

To get a better idea of how MFC and ATL versions of the same control compare, let's compare the two controls we've created in this chapter. Both controls provide similar functionality and features, including things such as the property page and events. As previously mentioned, both controls behave exactly the same within the limits of the specification we created earlier. However, control appearance isn't everything. The "What's Missing" section earlier shows you that these two controls aren't precisely the same, at least not at a low-level perspective. However, there are still other differences that we need to discuss from a developer's perspective.

First, let's review the coding part of the picture. It takes nearly four times longer to create the ATL version of the control. The main problem is that ATL doesn't provide anything in the way of automation. The developer creates every element of an ATL control; MFC, on the other hand, makes assumptions of what you want. There isn't anything wrong with either approach — it comes down to a question of flexibility versus development time.

Discover another developer issue by taking a quick look at all the code required to create the ATL version of the control in this chapter. Not only does this reinforce the developer time issue, but it should also tell you something about debugging time for an ATL-based control. Since you're writing all of the code, you also get to perform all of the debugging. (Unfortunately, some developers claim this is actually an advantage because of the number of bugs and inconsistencies in the current set of MFC libraries.)

The ATL-based control itself is also quite a bit larger at 92KB in comparison to the MFC-based version at 40 KB. However, appearances can be deceiving. Figure 13-37 shows the Dependency Walker views for both controls. Notice that the MFC-based control requires two additional support files: MFC42.DLL and MSVCRT.DLL, both of which are quite large.

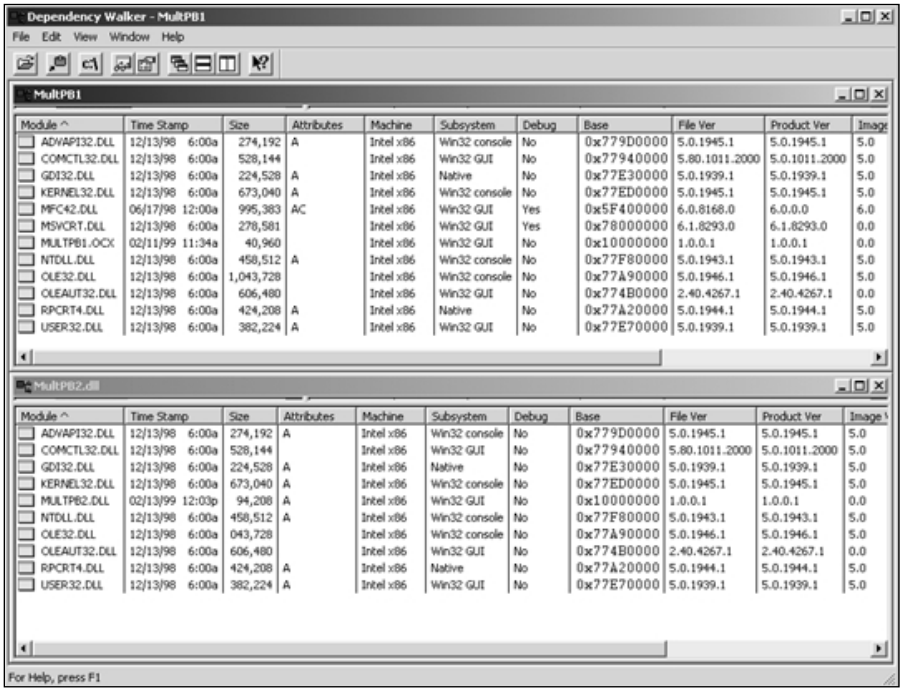


Figure 13-37: ATL-based ActiveX controls are self-contained; the MFC-based control requires support files.

If you can be absolutely certain that the user has the correct version of both of these support files, then the MFC-based ActiveX control is actually a better way to go from several standpoints. The most important consideration is that you can provide a smaller control with your application — a very important consideration if

users need to download your control. Obviously, Microsoft will also be responsible for upgrading the MFC libraries as well, which means that your control can benefit from automatic bug fixes as long as the MFC libraries remain compatible with your application.

However, it's easy to see why some developers are a little less than thrilled about using anything other than ATL to develop their ActiveX controls. Let's look at the issue of compatibility. There isn't any guarantee that the version of the MFC libraries on a user machine is the same as the ones that you use for development purposes. As a result, you can't guarantee the compatibility of MFC-based controls that you create unless you also control the user's environment. What happens if users lack the MFC libraries? They have to undergo a rather long download process, which is precisely the opposite of what you were trying to accomplish.

In sum, from a developer's perspective, ATL-based controls require a very large up-front investment, but are less prone to problems later. In addition, the developer is more familiar with the control and has more flexibility in determining which interfaces to include. On the other hand, MFC-based controls are much easier and faster to develop, but lack the flexibility and compatibility of an ATL-based control. Obviously, the technique you choose for developing application controls has to be based on your goals for that control.

Testing in an Application

Now that we have two versions of the same control to use, it's time to create an application in which to test them. Since the two controls work the same and use the same interfaces, we can use the same application to test them. As a result, you can find two versions of this application on the CD-ROM, even though only one of them is discussed in the text. The MPBTest application is used to test the MFC version of the control; the MPBTest2 application tests the ATL version of the control. The following sections show you how to build the test application.

CD-ROM Location: Chapt12\MPBTest and Chapt12\MPBTest2
Program Name: MPBTest.EXE and MPBTest2.EXE
Source Code Modules in Text: MultPB2.CPP, MultPB2.H, MultPB2.IDL

Creating a Program Shell

The first thing we need to do is create a program shell. We already created a dialog-based application in this book, so the following procedure is somewhat shorter than usual. Chapter 3 shows how to build a complete application in the "Dialog Based" section of the chapter.

Try This

1. Use the File | New command to display the New dialog box.
2. Highlight the MFC AppWizard (exe) icon on the Projects tab, type a project name in the Project name field (the example uses MPBTest), then click OK. Visual C++ shows you the MFC AppWizard - step 1 dialog box.
3. Choose the Dialog based option, and then click Finish. You should see a New Project Information dialog box similar to the one shown in Figure 13-38.

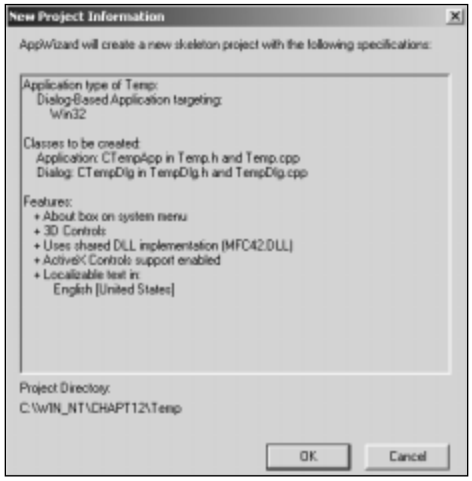


Figure 13-38: The New Project Information dialog box shows that this is a simple dialog-based application.

4. Click OK. Visual C++ creates the new project for you. Once the project shell is complete, you should see a new dialog box resource open. This is where we place the application controls. At this point, we're ready to add the one new component required to create this example — the Multiple State Pushbutton control. The control you choose in the steps that follow depends on which control you want to test.
5. Use the Project | Add to Project | Components and Controls command to display the Components and Controls Gallery dialog box shown in Figure 13-39. This is where you can choose new components to add to your application.
6. Double-click the Registered ActiveX Controls folder. Visual C++ displays a list of all of the controls installed on your machine.

This procedure assumes that you've registered the two Multiple State Pushbutton controls created in this chapter. The controls get registered automatically as part of the compile and link cycle. However, if you copy the controls directly from the CD-ROM, you need to use the RegSvr32 <Control Name> command to register the controls. Make sure you copy the controls to your hard drive before you attempt to register them.

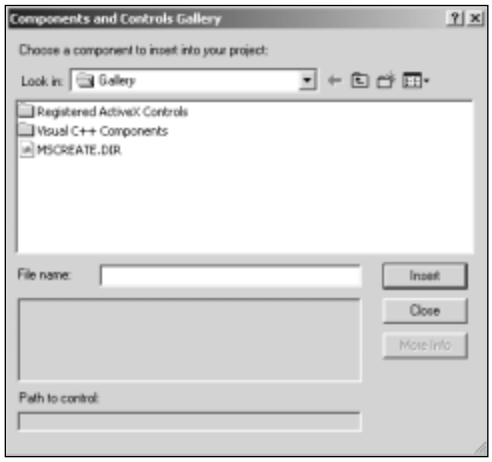


Figure 13-39: The Components and Controls Gallery dialog box shows you a list of components installed on your workstation.

7. Highlight one of the two Multiple State Pushbutton control entries, and then click Insert. You should see the Confirm Classed dialog box shown in Figure 13-40. This dialog box shows the class names and file names for each of the classes required to support the control you want to insert. In most cases, you won't need to change any of the information here unless there's a conflict with another control in your project.

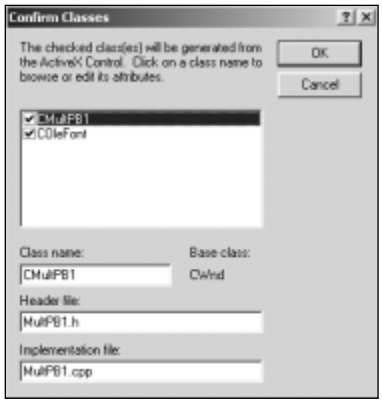


Figure 13-40: The Confirm Classes dialog box allows you to choose the class and file names associated with your control.

8. Click OK to accept the default names. Visual C++ adds the control to your project.

9. Click Close to close the Components and Controls Gallery dialog box.

Designing the Test Dialog

At this point, you have an application shell that contains a test dialog box and one of the two controls created earlier in the chapter. It's time to design a test dialog box that will allow you to test all of the capabilities of the test control. Figure 13-41 shows the controls that I've added to this example to test the functionality of the control. Table 13-4 shows the control settings for this example.

Figure 13-41: The test dialog box will allow you to test all of the capabilities of the Multiple State Pushbutton.

Table 13-4
Control Settings for the Multiple State Pushbutton Dialog Box

Control ID	*Property*	*Value*
IDC_MULTPB1CTRL1	ButtonType	0 - On/Off
IDC_ON_OFF	Caption	On/Off
	Group	Checked
	Tab stop	Checked
IDC_YES_NO	Caption	Red/Yellow/Green
	Tab stop	Checked

Control ID	*Property*	*Value*
IDC_RED_YELLOW_GREEN	Caption	Red/Yellow/Green
	Tab stop	Checked
IDC_USER_MODE	Caption	Red/Yellow/Green
	Tab stop	Checked
IDC_EDIT1	Disabled	Checked
IDC_DO_CHANGE	Caption	Do Change
IDC_DO_CLICK	Caption	Do Click

Part of the testing process for any control is to check the property pages to ensure they appear as expected. Make sure you check the two property pages that we added to the Multiple State Pushbutton control. Figure 13-42 shows the Visual C++ presentation of the Control tab, whereas Figure 13-43 shows the Font tab.

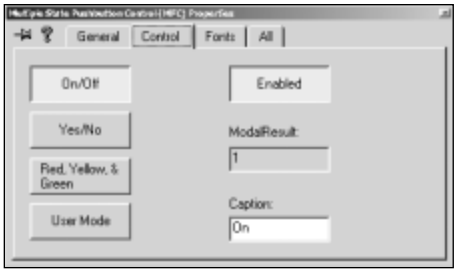


Figure 13-42: The Control tab of the Multiple State Pushbutton Control (MFC) Properties page looks much as you would expect it to.

Figure 13-43: Make sure you check presentations of any stock property pages in your control to ensure they work as intended.

It's interesting to compare differences in the presentation of stock property pages between ATL and MFC. Figure 13-43 shows the MFC version of the Font property page, whereas Figure 13-44 shows the ATL version. Yes, the two tabs work about the same, but there are differences in presentation that you need to be aware of.

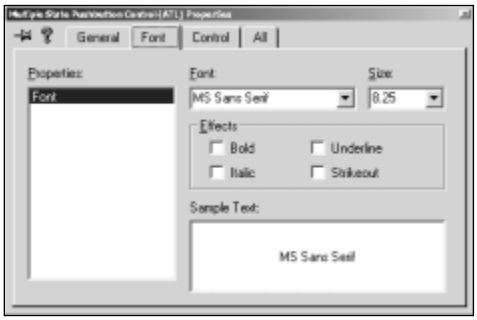


Figure 13-44: There are differences between the MFC and the ATL version of the Font tab.

Adding Some Code

It's time to add some code to the example. The first thing we need to do is add some member variables to access the various controls on the dialog box. There are two variables you need to add. The first member variable is a control member variable for IDC_EDIT1 named m_userModeString, the second is a control member variable for IDC_MULTPB1CTRL1 named m_multiButton. Chapter 3 covers the procedure for adding member variables, so I won't cover that process again here.

Once you have the member variables in place, you need to add event handlers for each of the radio buttons and pushbuttons. The only control without an associated event is IDC_EDIT1. Chapter 3 also covers this procedure, so I won't cover it again here.

There isn't any way to know in advance what the button state will be when we start the application. As a result, you need to add some initialization code that sets the correct option button on the dialog box. Otherwise, there isn't any way to determine that the right button caption is displayed when the application starts. Listing 13-24 shows the initialization code you need to add (in bold) to the OnInitDialog( ) method to perform this task.

Listing 13-24: **Test Application Dialog Box Initialization Code**

```
BOOL CMPBTestDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  // Add "About..." menu item to system menu.
  // IDM_ABOUTBOX must be in the system command range.
  ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
  ASSERT(IDM_ABOUTBOX < 0xF000);
  CMenu* pSysMenu = GetSystemMenu(FALSE);
  if (pSysMenu != NULL)
  {
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
```

```
    if (!strAboutMenu.IsEmpty())
    {
      pSysMenu->AppendMenu(MF_SEPARATOR);
      pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }
  }

  // Set the icon for this dialog.  The framework does this automatically
  //  when the application's main window is not a dialog
  SetIcon(m_hIcon, TRUE);      // Set big icon
  SetIcon(m_hIcon, FALSE);    // Set small icon

  // Determine which radio button to enable, then
  // enable it.
  switch (m_multiButton.GetButtonType())
  {
  case 0:
    CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_ON_OFF);
    break;
  case 1:
    CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_YES_NO);
    break;
  case 2:
    CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_RED_YELLOW_GREEN);
    break;
  case 3:
    CheckRadioButton(IDC_ON_OFF, IDC_USER_MODE, IDC_USER_MODE);
    m_userModeString.EnableWindow(TRUE);
    break;
  }

  return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Finishing the test code is a matter of filling in the event handlers. Each of these handlers tests the Multiple State Pushbutton control in some way — either by changing a property or by executing a method. Listing 13-25 contains the code you need to make this test program functional.

Listing 13-25: **Test Application Event Handler Source Code**

```
void CMPBTestDlg::OnClickMultpb1ctrl1()
{
  // Determine which button was pressed, then
  // display an appropriate message.
  switch (m_multiButton.GetModalResult())
```

Continued

Listing 13-25: *(continued)*

```
  {
  case -1:
    AfxMessageBox("-1 - User Mode", MB_OK);
    break;
  case 1:
    AfxMessageBox("1 - On", MB_OK);
    break;
  case 2:
    AfxMessageBox("2 - Off", MB_OK);
    break;
  case 3:
    AfxMessageBox("3 - Yes", MB_OK);
    break;
  case 4:
    AfxMessageBox("4 - No", MB_OK);
    break;
  case 5:
    AfxMessageBox("5 - Red", MB_OK);
    break;
  case 6:
    AfxMessageBox("6 - Yellow", MB_OK);
    break;
  case 7:
    AfxMessageBox("7 - Green", MB_OK);
    break;
  }
}

void CMPBTestDlg::OnOnOff()
{
  // Change the button type to On/Off.
  m_multiButton.SetButtonType(0);

  // Disable the user mode text box.
  m_userModeString.EnableWindow(FALSE);
}

void CMPBTestDlg::OnYesNo()
{
  // Change the button type to Yes/No.
  m_multiButton.SetButtonType(1);

  // Disable the user mode text box.
  m_userModeString.EnableWindow(FALSE);
}

void CMPBTestDlg::OnRedYellowGreen()
{
```

```
  // Change the button type to Red/Yellow/Green.
  m_multiButton.SetButtonType(2);
  // Disable the user mode text box.
  m_userModeString.EnableWindow(FALSE);
}

void CMPBTestDlg::OnUserMode()
{
  // Change the button type to User Mode.
  m_multiButton.SetButtonType(3);

  // Enable the user mode text box.
  m_userModeString.EnableWindow(TRUE);
}

void CMPBTestDlg::OnChangeEdit1()
{
  CString oTextValue;
  // Get the current user mode string value.
  m_userModeString.GetWindowText(oTextValue);

  // Set the user mode string value.
  m_multiButton.SetUser(oTextValue.GetBuffer(80));
}

void CMPBTestDlg::OnDoChange()
{
  // Go to the next button value without generating
  // an OnClick event.
  m_multiButton.DoChange();
}

void CMPBTestDlg::OnDoClick()
{
  // Go to the next button value and generate an OnClick event.
  m_multiButton.DoClick();
}
```

As you can see, the code for this example is very simple. In fact, simplicity in your test application is something to strive for to ensure that you're not trying to fix bugs in your test application while you look for bugs in a control. In most cases, the various event handlers change a single property or query a single method in the control. The radio button event handlers also have to track the current state of the IDC_EDIT1 control to ensure that it's enabled only in user mode. Finally, the OnClickMultpb1ctrl1( ) method reacts to click events output by the control. This method simply displays a message box that shows the current ModalResult value.

Summary

This chapter provides you with the practical knowledge required to determine which kind of component technology to use, then implement that technology in the form of a new component. As a follow-up to Chapter 12, this chapter concentrates on the kinds of components that you use to create a desktop or end-user Web-based application. In fact, our discussion is limited to only the two most popular kinds of desktop component technologies: ATL and MFC. Here are some additional conclusions you can draw from the contents of this chapter.

- The Lite versions of the ATL-based ActiveX controls are designed exclusively for use by Internet Explorer and shouldn't be used with desktop applications.
- You can use the RegSvr32 -U <ActiveX Control Filename> command to unregister an ActiveX control prior to removing it from your machine. Always make sure that you perform this step prior to removing a control to ensure the associated registry entries are removed. Doing so reduces the probability of machine lockups and registry bloat.
- Both the MFC and ALT methods of creating an ActiveX control allow you to subclass an existing Windows control, which reduces the amount of time required to develop a new control. For example, creating custom pushbuttons is much easier if you subclass the Windows version of the button class first.
- A combination of an About dialog box and version information for your control ensures that other programmers can contact you as needed for help in using a custom control. The version information also helps you protect your copyright on any controls you create.
- An ActiveX control automatically takes care of the implementation of any stock properties, methods, and events for you. However, you need to override the default implementation if your control needs to perform any special processing of these stock options. On the other hand, custom properties, methods, and events are unique to your control and always require additional code to ensure persistence, connectivity, and functionality.
- ActiveX controls normally use the BSTR, not a C/C++ string type to transfer text information to and from the client. However, Visual C++ automatically adds code to translate the incoming text to a C class for you when working with MFC.
- There are two standard sizes for property pages: 250 X 62 pixels and 250 X 110 pixels.
- Microsoft provides stock property pages that you can use for font- and color-related properties.
- You use a combination of the OnGetDisplayString( ), OnGetPredefinedStrings( ), and OnGetPredefinedValue( ) methods to display a selection list for properties that have a limited number of discrete input values.

- Always use CoTaskMemAlloc( ) to allocate memory in a component rather than malloc( ).
- Two controls must provide exactly the same interfaces to provide the same services and support.

✦ ✦ ✦